Гради Буч

Объектно-ориентированный анализ и проектирование
с примерами приложений на С++

ВТОРОЕ ИЗДАНИЕ

Rational Санта- Клара, Калифорния
перевод с английского под редакцией И. Романовского и Ф. Андреева

Оглавление

Об авторе

Гради Буч (Grady Booch), главный исследователь корпорации Rational Software, признан всем международным сообществом разработчиков программного обеспечения благодаря его основополагающим работам в области объектно-ориентированных методов и приложений. Он — постоянный автор в таких журналах, как «Object Magazine» и «C++ Report» и автор многих бестселлеров, посвященных объектно-ориентированному проектированию и разработке программ. Гради Буч редактирует и участвует в написании серии «Разработка объектно-ориентированного программного обеспечения» («Object-oriented Software Engineering Series»), издаваемой Addison-Wesley Longman.

Человечество, по милости Божией, жаждет душевного покоя, эстетических достижений, безопасности семьи, справедливости и свободы, не удовлетворяясь повышением производительности труда. Но именно повышение производительности труда позволяет делиться избыточным, а не драться за недостающее; оно освобождает время для духовного, эстетического и семейного. Оно позволяет обществу направить часть своих средств на институты религии, правосудия и сохранения свобод.

Арлан Миллс (Harlan Mills) DPMA и человеческая производительность (DPMA and Human Productivity)

Предисловие

Как программисты-профессионалы мы стремимся делать свои системы полезными и работоспособными. Как инженеры-разработчики мы должны создавать сложные системы при ограниченных компьютерных и кадровых ресурсах. За последние несколько лет объектно-ориентированная технология проникла в различные разделы компьютерных наук. К ней относятся как к средству преодоления сложности, присущей многим реальным системам. Объектная модель показала себя мощной объединяющей концепцией.

Что изменилось по сравнению с первым изданием

Со времени выхода в свет первого издания книги "Объектно-ориентированное проектирование с примерами применения" ("Object-Oriented Design with Applications") объектно-ориентированная технология стала одной из основных при разработке программного обеспечения промышленного масштаба. Мы видим, что во всем мире объектная парадигма применяется в таких различных областях, как управление банковскими транзакциями, автоматизация кегельбанов, управление коммунальным хозяйством и исследование генов человека. Во многих случаях новые поколения операционных систем, систем управления базами данных, телефонных служб, систем авионики и мультимедиа-программ пишутся в объектно-ориентированном стиле. В большинстве таких проектов предпочли использовать объектно-ориентированную технологию просто потому, что не было другой возможности создать достаточно надежную и жизнеспособную систему.

За последние годы в сотнях проектов применяли нотацию и процесс разработки, предложенные в нашей книге.[1] В процессе собственной разработки проектов и с учетом опыта многих других, кто пожертвовал своим временем, чтобы поделиться с нами, мы нашли много способов усовершенствовать наш метод. Усовершенствование достигается за счет лучшего изложения процесса проектирования, введения семантики, которая ранее не была отражена в нашей нотации, и упрощения этой нотации там, где возможно.

За истекшее время появились многие другие методы, изложенные в работах Джекобсона (Jacobson), Румбаха (Rumbaugh), Гоада и Иордана (Goad and Yourdon), Константайна (Constantine), Шлера и Меллора (Shiaer and Mellor), Мартина и Одел-ла (Martin and Odell), Вассермана (Wasserman), Голдберга и Рубина (Goldberg and Rubin), Эмбли (Embley), Вирфс-Брока (Wirfs-Brock), Голдстейна и Алгера (Goldstein and Alger), Хендерсон-Селлерса (Henderson-Sellers), Файесмита (Firesmith) и др. Особенно интересна работа Румбаха, который отмечает, что в наших подходах больше сходства чем различий. Мы провели анализ многих из этих методов, разговаривали с разработчиками и менеджерами, которые их использовали, и, когда это было возможно, пытались сами их применять. Так как мы больше заинтересованы в реальной помощи по разработке проектов в объектно-ориентированной технологии, чем в догматическом следовании (будь то по эмоциональным или историческим причинам) нашим идеям, мы пытались включить все лучшее, что нашли в новых методах, в нашу собственную работу. Мы с благодарностью отмечаем фундаментальный и уникальный вклад каждого из этих лиц в данную область.

Индустрии программных средств и объектно-ориентированной технологии полезно было бы иметь, в частности, стандартную систему обозначений. Поэтому в

[1] Включая мои собственные проекты. Я все же разработчик, а не методолог. Первый вопрос, который нужно задавать каждому методологу: "Используете ли вы ваши методы при разработке собственных программ?".

данном издании представлена унифицированная система обозначений, в которой, где возможно, устранены косметические различия между нашей нотацией и другими, особенно Джекобсона и Румбаха. Как и раньше, чтобы обеспечить ее неограниченное использование, система обозначений сделана общедоступным продуктом (public domain).

Цели, аудитория и структура этого издания остаются теми же, что и в первом. Однако, есть пять существенных различий между вторым и первым изданиями.

Во-первых, глава 5 была расширена с тем, чтобы изложить унифицированную систему обозначений значительно подробнее. Чтобы сделать ее более понятной, были явно разделены существенные и дополнительные элементы обозначений. Кроме того, особое внимание уделено взаимному согласованию разных представлений в этой системе.

Во-вторых, значительно расширены главы 6 и 7, в которых рассматривается практика объектно-ориентированного анализа и проектирования. Мы даже сменили в этом издании заглавие книги, отразив тот факт, что наш метод объединяет анализ и проектирование.

В-третьих, мы решили приводить примеры всех программных текстов в основной части книги на одном языке, а именно на C++. Этот язык быстро становится фактическим стандартом для многих областей, кроме того, большинство профессиональных разработчиков, "сочиняющих" на других языках, могут "читать" на C++. Это не значит, что мы считаем другие языки - такие, как Smalltalk, CLOS, Ada или Eiffel- менее важными. Главная цель этой книги - анализ и проектирование, и так как нам нужны конкретные примеры, мы решили писать их на достаточно общем языке программирования. Где возможно, мы описываем особенности семантики других языков и их влияние на наш метод.

В-четвертых, в это издание включены несколько новых примеров приложений. Некоторые интересные идиомы и среды разработки применялись для написания целого ряда приложений и наши примеры отражают эти достижения. Например, концепция "клиент/сервер" послужила основой для одного пересмотренного прикладного примера.

Наконец, почти в каждую главу добавлены ссылки на литературу. Кроме того, мы обсуждаем новые проблемные вопросы объектно-ориентированной технологии, возникшие после выхода первого издания.

Цели

Эта книга призвана служить практическим руководством по созданию объектно-ориентированных систем. Особое внимание мы уделяем следующим целям:

- обеспечить отчетливое понимание основных концепций объектной модели;
- помочь освоить систему обозначений и процесс объектно-ориентированного анализа и проектирования;
- научить читателя практическому применению объектно-ориентированного подхода в различных предметных областях.

Изложенные здесь понятия имеют серьезное теоретическое обоснование, но эта книга прежде всего призвана удовлетворить практические потребности и интересы сообщества разработчиков программных продуктов.

Аудитория

Книга предназначена и для профессионалов, и для студентов:

- Разработчику-практику мы покажем, как эффективно применять объектно-ориентированную технологию для решения реальных задач.
- Если вы выступаете в роли аналитика или архитектора системы, мы поможем вам пройти путь от постановки задачидо реализации, с использованием объектно-ориентированного анализа и проектирования. Мы разовьем вашу способность отличать "хорошую" объектно-ориентированную архитектуру от "плохой" и находить правильное решение в сложном реальном мире. Возможно самое важное, что мы предлагаем - новые подходы к рассмотрению сложных систем.
- Менеджеру программного проекта мы подскажем, как распределить ресурсы в команде разработчиков и снизить издержки, связанные с написанием любой сложной программной системы.
- Создателю инструментальных программных средств и их пользователю мы предложим подробное изложение системы обозначений и процесса объектно-ориентированной разработки - основы CASE (computer-aided software engineering, разработка программ с помощью компьютера).
- Студенту книга будет полезна, как основа, которая поможет приобрести начальные знания и навыки в искусстве создания сложных систем.

Книга может быть использована при чтении курсов для студентов и аспирантов, а также при проведении профессиональных семинаров и самостоятельном изучении. Так как она посвящена в основном методу построения программ, книга идеально подойдет для курсов проектирования программных продуктов и даст материал для дополнительных занятий по курсам объектно-ориентированных языков.

Структура

Книга делится на три большие части - "Концепции", "Метод" и "Примеры приложений" - с добавлением значительного дополнительного материала.

Концепции

Первая часть посвящена анализу сложности, присущей программным системам, в частности анализу того, как эта сложность проявляется. Мы вводим объектную модель как средство борьбы со сложностью. Мы рассматриваем основные элементы объектной модели: абстрагирование, инкапсуляцию, модульность, иерархию, типизацию, параллелизм, устойчивость. Мы задаемся такими глубинным вопросами как "Что такое класс?" и "Что такое объект?". Поскольку выявление осмысленных классов и объектов - ключевая задача объектно-ориентированного проектирования, значительное время мы уделяем вопросам классификации. В частности, мы рассматриваем подходы к классификации в других дисциплинах: биологии, лингвистике и психологии, а затем применяем полученные выводы к обнаружению классов и объектов внутри программных систем.

Метод

Вторая часть описывает метод построения сложных систем, основанный на объектной модели. Сначала мы вводим систему графических обозначений объектно-ориентированного анализа и проектирования, а затем рассматриваем процесс разработки. Мы затрагиваем и практические вопросы, в частности роль этого процесса в жизненном цикле программного продукта и его значение для управления проектами.

Примеры приложений

Заключительная часть посвящена пяти нетривиальным примерам, охватывающим широкий круг приложений: сбору данных, прикладным средам разработки, архитектуре клиент/сервер, искусственному интеллекту и управлению технической системой. Мы выбрали эти области, так как они хорошо представляют те разновидности сложных задач, с которыми может столкнуться программист. Легко можно продемонстрировать успех любых принципов на простых задачах, но поскольку мы фокусируем свое внимание на создании систем реальной жизни, нам было интереснее показать, как объектная модель доходит до сложных приложений. Некоторые читатели могут быть незнакомы со спецификой выбранного приложения, поэтому мы начинаем каждый пример с краткого обсуждения присущих ему технологических особенностей (таких, как проектирование базы данных и понятия информационной доски). Разработку программных систем нельзя свести к набору рецептов, поэтому мы подчеркиваем необходимость постепенного развития приложений на основе соблюдения ряда четких принципов и следования ясным моделям.

Дополнительный материал

В текст книги вплетен значительный дополнительный материал. В большинстве глав имеются специальные вставки (врезки), в которых содержится информация по отдельным важным темам, например, о механизмах вызова методов в различных объектно-ориентированных языках программирования. В книгу включено также приложение, посвященное объектно-ориентированным языкам, в котором рассматривается различие между объектными и объектно-ориентированными языками, их эволюция и свойства. Для тех читателей, которые незнакомы с конкретными языками программирования, мы подготовили сводку свойств нескольких основных языков с примерами кода. В книге имеется глоссарий (словарь основных терминов) и обширная тематическая библиография. Наконец, на последних страницах содержится сводка по объектно-ориентированному методу разработки и системе обозначений.

Помимо этой книги, можно порекомендовать "Сборник задач", содержащий упражнения, вопросы и проекты, которые должны оказаться полезными для семинарских занятий. "Сборник задач" ("Instructor's Guide with Exercises", ISBN 0-8053-5341-0) написан Мэри Бет Россон (Mary Beth Rosson) из лаборатории Томаса Дж. Ватсона (Thomas J. Watson) корпорации IBM. Преподаватели, желающие получить эту книгу, могут обращаться за бесплатным экземпляром непосредственно в издательство Addison-Wesley Longman (aw.cse@aw.com) или к местному представителю этого издательства. Вопросы и предложения для сборника задач можно направлять по адресу: rosson@watson.ibm.com.

Приобрести инструментальные средства и пройти обучение методу Буча (Booch) можно в разных местах. За дополнительной информацией обращайтесь в компанию Rational: booch-card@rational.com. Кроме того, Addison-Wesley Longman может предоставить учебным заведениям программные средства, поддерживающие нашу нотацию.

Как пользоваться этой книгой?

Книгу можно читать от корки до корки, но можно и по-другому. Если вы нуждаетесь в глубоком понимании объектной концепции и принципов объектно-ориентированного проектирования, начните с главы 1 и следуйте далее по порядку. Если вам интересна в основном система обозначений и процесс объектно-ориентированного анализа и проектирования, начните с глав 5 и 6; менеджерам проектов, использующим этот метод, будет особенно интересна глава 7. Если вы интересуетесь практическим приложением объектно-ориентированной технологии к конкретной области, обратитесь к главам 8-12.

Благодарности

Книга посвящается моей жене в благодарность за ее любовь и поддержку.

На протяжении всей работы над первым и вторым изданиями много людей формировали мои взгляды на объектно-ориентированную разработку. Среди них были: Сэм Адаме (Sam Adams), Майк Акроид (Mike Akroid), Гленн Андерт (Glenn Andert), Сид Байлин (Sid Bailin), Кент Бек (Kent Beck), Даниел Бобров (Daniel Bobrow), Дик Больц (Dick Bolz), Дэйв Балман (Dave Bulman), Дэйв Бернстейн (Dave Bernstein), Кэйван Кэран (Kayvan Carun), Дэйв Коллинз (Dave Collins), Стив Кук (Steve Cook), Дамиан Конвэй (Damian Conway), Джим Коплиен (Jim Coplien), Брэд Кокс (Brad Cox), Ворд Канингэм (Ward Cunningham), Том ДеМар-ко (Torn DeMarco), МайкДелвин (Mike Delvin), Ричард Габриел (Richard Gabriel), Вильям Ценемерас (William Cenemeras), Адель Голдберг (Adele Goldberg), Ян Грэ-хем (lan Graham), Тони Хоар (Топу Ноаге), Джон Хопкинс (Jon Hopkins), Майкл Джэксон (Michael Jackson), Ральф Джонсон (Ralph Johnson), Джеймс Кемпф (James Kempf). Норм Керт (Norm Kerth), Иордан Крейндлер (Jordan Kreindler), Дуг Ли (Doug Lea), Фил Леви (Phil Levy), Барбара Лисков (Barbara Liskov), Клифф Лонгмэн (Cliff Longman), Джеймс МакФарлэйн (James MacFarlane), Масауд Ми-лани (Masoud Milani), Арлан Миллс (Harlan Mills), Роберт Мюррей (Robert Murray), Стив Нейс (Steve Neis), Джин Уйе (Gene Ouye), Дэйв Парнас (Dave Parnas), Билл Риддел (Bill Riddel), Мэри Бет Россон (Mary Beth Rosson), Кенни Рубин (Кеппу Rubin), Джим Румбах (Jim Rumbaugh), Курт Шмукер (Kurt Schmucker), Эд Сейде-витц (Ed Seidewitz), Дэн Шифман (Dan Shiftman), Дэйв Стивенсон (Dave Stevenson), Бьерн Страуструп (Bjarne Stroustrup), Дэйв Томсон (Dave Thomson), Майк Вило (Mike Vilot), Тони Вассерман (Tony Wasserman), Питер Вегнер (Peter Wegner), Айсеал Байт (Iseult White), Джон Вильяме (John Williams), Ллойд Вильяме (Lloyd Williams), Марио Волчко (Mario Wolczko), Никлаус Вирт (Niklaus Wirth) и Эд Иордан (Ed Yourdon).

Практические главы этой книги формировались по мере моего участия в разработке сложных программных систем по всему миру для таких компаний как: Apple, Alcatel, Andersen Consulting, AT&T, Autotrol, Bell Northern Research, Boeing, Borland, Computer Sciences Corporation, Contel, Ericsson, Ferranti, General Electric, GTE, Holland Signaal, Hughes Aircraft Company, IBM, Lockheed, Martin Marietta, Motorola, NTT, Philips, Rockwell International, Shell Oil, Symantec, Taligent и TRW. Я общался с сотнями профессиональных программистов и менеджеров и благодарю их всех за то, что они помогли сделать эту книгу отвечающей проблемам реальной жизни.

Особая благодарность - компании Rational за поддержку моего труда. Спасибо также моему редактору Дэну Йоранстаду (DanJoraanstad) за его постоянную поддержку и Тони Холлу (Tony Hall), рисунки которого внесли жизнь в то, что без них осталось бы еще одной скучной технической книгой. Наконец, спасибо трем моим кошкам, Кэми (Сату), Энни (Annie) и Тени (Shadow), составлявшим мне компанию в долгие часы ночной работы.

ЧАСТЬ ПЕРВАЯ

Концепции

Сэр Исаак Ньютон по секрету признавался друзьям, что он знает, ***как*** гравитация ведет себя, но не знает, ***почему***.

Лили Томлин (Lily Tomlin)
В поисках признаков разумной жизни во Вселенной
(The Search for Signs of Intelligent Life in the Universe)

Глава 1

Сложность

Врач, строитель и программистка спорили о том, чья профессия древнее. Врач заметил: «В Библии сказано, что Бог сотворил Еву из ребра Адама. Такая операция может быть проведена только хирургом, поэтому я по праву могу утверждать, что моя профессия самая древняя в мире». Тут вмешался строитель и сказал: «Но еще раньше в Книге Бытия сказано, что Бог сотворил из хаоса небо и землю. Это было первое и, несомненно, наиболее выдающееся строительство. Поэтому, дорогой доктор, вы не правы. Моя профессия самая древняя в мире». Программистка при этих словах откинулась в кресле и с улыбкой произнесла: «А кто же по-вашему сотворил хаос?»

1.1. Сложность, присущая программному обеспечению

Простые и сложные программные системы

Звезда в преддверии коллапса; ребенок, который учится читать; клетки крови, атакующие вирус, — это только некоторые из потрясающе сложных объектов физического мира. Компьютерные программы тоже бывают сложными, однако их сложность совершенно другого рода. Брукс пишет: «Эйнштейн утверждал, что должны существовать простые объяснения природных процессов, так как Бог не действует из каприза или по произволу. У программиста нет такого утешения: сложность, с которой он должен справиться, лежит в самой природе системы» [I].

Мы знаем, что не все программные системы сложны. Существует множество программ, которые задумываются, разрабатываются, сопровождаются и используются одним и тем же человеком. Обычно это начинающий программист или профессионал, работающий изолированно. Мы не хотим сказать, что все такие системы плохо сделаны или, тем более, усомниться в квалификации их создателей. Но такие системы, как правило, имеют очень ограниченную область применения и короткое время жизни. Обычно их лучше заменить новыми, чем пытаться повторно использовать, переделывать или расширять. Разработка подобных программ скорее утомительна, чем сложна, так что изучение этого процесса нас не интересует.

Нас интересует разработка того, что мы будем называть *промышленными программными продуктами.* Они применяются для решения самых разных задач, таких, например, как системы с обратной связью, которые управляют или сами управляются событиями физического мира и для которых ресурсы времени и памяти ограничены; задачи поддержания целостности информации объемом в сотни ты-сяч записей при параллельном доступе к ней с обновлениями и запросами; системы управления и контроля за реальными процессами (например, диспетчеризация воздушного или железнодорожного транспорта). Системы подобного типа обычно имеют большое время жизни, и большое количество пользователей оказывается в зависимости от их нормального функционирования. В мире промышленных программ мы также встречаем среды разработки, которые упрощают создание приложений в конкретных областях, и программы, которые имитируют определенные стороны человеческого интеллекта.

Существенная черта промышленной программы — уровень сложности: один разработчик практически не в состоянии охватить все аспекты такой системы. Грубо говоря, сложность промышленных программ превышает возможности человеческого интеллекта. Увы, но сложность, о которой мы говорим, по-видимому, присуща всем большим программных системам. Говоря *«присуща»,* мы имеем в виду, что эта сложность здесь неизбежна: с ней можно справиться, но избавиться от нее нельзя.

Конечно, среди нас всегда есть гении, которые в одиночку могут выполнить работу группы обычных людей-разработчиков и добиться в своей области успеха, сравнимого с достижениями Франка Ллойда Райта или Леонардо да Винчи. Такие люди нам нужны как

архитекторы, которые изобретают новые идиомы, механизмы и основные идеи, используемые затем при разработке других систем. Однако, как замечает Петере: «В мире очень мало гениев, и не надо думать, будто в среде программистов их доля выше средней» [2]. Несмотря на то, что все мы чуточку гениальны, в промышленном программировании нельзя постоянно полагаться на божественное вдохновение, которое обязательно поможет нам. Поэтому мы должны рассмотреть более надежные способы конструирования сложных систем. Для лучшего понимания того, чем мы собираемся управлять, сначала ответим на вопрос: почему сложность присуща всем большим программным системам?

Почему программному обеспечению присуща сложность?

Как говорит Брукс, «сложность программного обеспечения — отнюдь не случайное его свойство» [3]. Сложность вызывается четырьмя основными причинами: сложностью реальной предметной области, из которой исходит заказ на разработку;

трудностью управления процессом разработки; необходимостью обеспечить дос-таточную гибкость программы; неудовлетворительными способами описания поведения больших дискретных систем.

Сложность реального мира. Проблемы, которые мы пытаемся решить с помощью программного обеспечения, часто неизбежно содержат сложные элементы, а к соответствующим программам предъявляется множество различных, порой взаимоисключающих требований. Рассмотрим необходимые характеристики электронной системы многомоторного самолета, сотовой телефонной коммутаторной системы и робота. Достаточно трудно понять, даже в общих чертах, как работает каждая такая система. Теперь прибавьте к этому дополнительные требования (часто не формулируемые явно), такие как удобство, производительность, стоимость, выживае-мость и надежность! Сложность задачи'и порождает ту сложность программного продукта, о которой пишет Брукс.

Эта внешняя сложность обычно возникает из-за «нестыковки» между пользователями системы и ее разработчиками: пользователи с трудом могут объяснить в форме, понятной разработчикам, что на самом деле нужно сделать. Бывают случаи, когда пользователь лишь смутно представляет, что ему нужно от будущей программной системы. Это в основном происходит не из-за ошибок с той или иной стороны; просто каждая из групп специализируется в своей области, и ей недостает знаний партнера. У пользователей и разработчиков разные взгляды на сущность проблемы, и они делают различные выводы о возможных путях ее решения. На самом деле, даже если пользователь точно знает, что ему нужно, мы с трудом можем однозначно зафиксировать все его требования. Обычно они отражены на многих страницах текста, «разбавленных» немногими рисунками. Такие документы трудно поддаются пониманию, они открыты для различных интерпретаций и часто содержат элементы, относящиеся скорее к дизайну, чем к необходимым требованиям разработки.

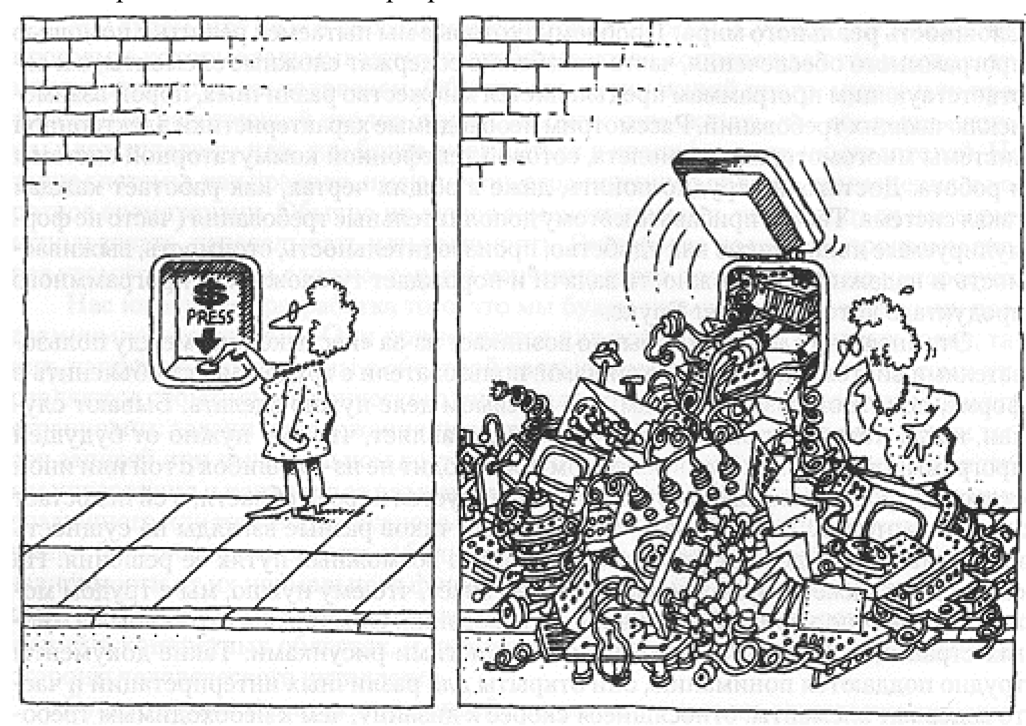
Дополнительные сложности возникают в результате изменений требований к программной системе уже в процессе разработки. В основном требования корректируются из-за того, что само осуществление программного проекта часто изменяет проблему. Рассмотрение первых результатов — схем, прототипов, — и использование системы после того, как она разработана и установлена, заставляют пользователей лучше понять и отчетливей сформулировать то, что им действительно нужно. В то же время этот процесс повышает квалификацию разработчиков в предметной области и позволяет им задавать более осмысленные вопросы, которые проясняют темные места в проектируемой системе.

Большая программная система — это крупное капиталовложение, и мы не можем позволить себе выкидывать сделанное при каждом изменении внешних требований. Тем не менее даже большие системы имеют тенденцию к эволюции в процессе их использования: следовательно, встает задача о том, что часто неправильно называют *сопровождением программного обеспечения.* Чтобы быть более точными, введем несколько терминов: под *сопровождением* понимается устранение ошибок;

под *эволюцией* — внесение изменений в систему в ответ на изменившиеся требования к ней; под *сохранением* — использование всех возможных и невозможных способов для

поддержания жизни в дряхлой и распадающейся на части системе. К сожалению, опыт показывает, что существенный процент затрат на разработку программных систем тратится именно на сохранение.

Трудности управления процессом разработки. Основная задача разработчиков состоит в создании иллюзии простоты, в защите пользователей от сложности описываемого предмета или процесса. Размер исходных текстов программной системы



Задача разработчиков программной системы — создать иллюзию простоты

отнюдь не входит в число ее главных достоинств, поэтому мы стараемся делать исходные тексты более компактными, изобретая хитроумные и мощные методы, а также используя среды разработки уже существующих проектов и программ. Однако новые требования для каждой новой системы неизбежны, а они приводят к необходимости либо создавать много программ «с нуля», либо пытаться по-новому использовать существующие. Всего 20 лет назад программы объемом в несколько тысяч строк на ассемблере выходили за пределы наших возможностей. Сегодня обычными стали программные системы, размер которых исчисляется десятками тысяч или даже миллионами строк на языках высокого уровня. Ни один человек никогда не сможет полностью понять такую систему. Даже если мы правильно разложим ее на составные части, мы все равно получим сотни, а иногда и тысячи отдельных модулей. Поэтому такой объем работ потребует привлечения команды разработчиков, в идеале как можно меньшей по численности. Но какой бы она ни была, всегда будут возникать значительные трудности, связанные с организацией коллективной разработки. Чем больше разработчиков, тем сложнее связи между ними и тем сложнее координация, особенно если участники работ географически удалены друг от друга, что типично в случае очень больших проектов. Таким образом, при коллективном выполнении проекта главной задачей руководства является поддержание единства и целостности разработки.

Гибкость программного обеспечения. Домостроительная компания обычно не имеет собственного лесхоза, который бы ей поставлял лес для пиломатериалов; совершенно необычно, чтобы монтажная фирма соорудила свой завод для изготовления стальных балок под будущее здание. Однако в программной индустрии такая практика — дело обычное. Программирование обладает предельной гибкостью, и разработчик может сам обеспечить себя всеми необходимыми элементами, относящимися к любому уровню абстракции. Такая гибкость чрезвычайно соблазнительна. Она заставляет разработчика создавать своими силами все базовые строительные блоки

будущей конструкции, из которых составляются элементы более высоких уровней абстракции. В отличие от строительной индустрии, где существуют единые стандарты на многие конструктивные элементы и качество материалов, в программной индустрии таких стандартов почти нет. Поэтому программные разработки остаются очень трудоемким делом.

Проблема описания поведения больших дискретных систем. Когда мы кидаем вверх мяч, мы можем достоверно предсказать его траекторию, потому что знаем, что в нормальных условиях здесь действуют известные физические законы. Мы бы очень удивились, если бы, кинув мяч с чуть большей скоростью, увидели, что он на середине пути неожиданно остановился и резко изменил направление движения.[2] В недостаточно отлаженной программе моделирования полета мяча такая ситуация легко может возникнуть.

Внутри большой прикладной программы могут существовать сотни и даже тысячи переменных и несколько потоков управления. Полный набор этих переменных, их текущих значений, текущего адреса и стека вызова для каждого процесса описывает состояние прикладной программы в каждый момент времени. Так как исполнение нашей программы осуществляется на цифровом компьютере, мы имеем систему с дискретными состояниями. Аналоговые системы, такие, как движение брошенного мяча, напротив, являются непрерывными. Д. Парнас [4] пишет: «когда мы говорим, что система описывается непрерывной функцией, мы имеем ввиду, что в ней нет скрытых сюрпризов. Небольшие изменения входных параметров всегда вызовут небольшие изменения выходных». С другой стороны, дискретные системы по самой своей природе имеют конечное число возможных состояний, хотя в больших системах это число в соответствии с правилами комбинаторики очень велико. Мы стараемся проектировать системы, разделяя их на части так, чтобы одна часть минимально воздействовало на другую. Однако переходы между дискретными состояниями не могут моделироваться непрерывными функциями. Каждое событие, внешнее по отношению к программной системе, может перевести ее в новое состояние, и, более того, переход из одного состояния в другое не всегда детерминирован. При неблагоприятных условиях внешнее событие может нарушить текущее состояние системы из-за того, что ее создатели не смогли предусмотреть все возможные варианты. Представим себе пассажирский самолет, в котором система управления полетом и система электроснабжения объединены. Было бы очень неприятно, если бы от включения пассажиром, сидящим на месте 38J, индивидуального освещения самолет немедленно вошел бы в глубокое пике. В непрерывных системах такое поведение было бы невозможным, но в дискретных системах любое внешнее событие может повлиять на любую часть внутреннего состояния системы. Это, очевидно, и является главной причиной обязательного тестирования наших систем; но дело в том, что за исключением самых тривиальных случаев, всеобъемлющее тестирование таких программ провести невозможно. И пока у нас нет ни математических инструментов, ни интеллектуальных возможностей для полного моделирования поведения больших дискретных систем, мы должны удовлетвориться разумным уровнем уверенности в их правильности.

Последствия неограниченной сложности

«Чем сложнее система, тем легче ее полностью развалить» [5]. Строитель едва ли согласится расширить фундамент уже построенного 100-этажного здания. Это не просто дорого: делать такие вещи значит напрашиваться на неприятности. Но что удивительно, пользователи программных систем, не задумываясь, ставят подобные задачи перед разработчиками. Это, утверждают они, всего лишь технический вопрос для программистов.

---

[2] Даже простые непрерывные системы могут иметь сложное поведение ввиду наличия хаоса. Хаос привносит случайность, исключающую точное предсказание будущего состояния системы. Например, зная начальное положение двух капель воды в потоке, мы не можем точно предсказать, на каком расстоянии друг от друга они окажутся по прошествии некоторого времени. Хаос проявляется в таких различных системах, как атмосферные процессы, химические реакции, биологические системы и даже компьютерные сети. К счастью, скрытый порядок, по-видимому, есть во всех хаотических системах, в виде так называемых *аттракторов.*

Наше неумение создавать сложные программные системы проявляется в проектах, которые выходят за рамки установленных сроков и бюджетов и к тому же не соответствуют начальным требованиям. Мы часто называем это *кризисом программного обеспечения,* но, честно говоря, недомогание, которое тянется так долго, становится нормой. К сожалению, этот кризис приводит к разбазариванию человеческих ресурсов — самого драгоценного товара — и к существенному ограничению возможностей создания новых продуктов. Сейчас просто не хватает хороших программистов, чтобы обеспечить всех пользователей нужными программами. Более того, существенный процент персонала, занятого разработками, в любой организации часто должен заниматься сопровождением и сохранением устаревших программ. С учетом прямого и косвенного вклада индустрии программного обеспечения в развитие экономики большинства ведущих стран, нельзя позволить, чтобы существующая ситуация осталась без изменений.

Как мы можем изменить положение дел? Так как проблема возникает в результате сложности структуры программных продуктов, мы предлагаем сначала рассмотреть способы работы со сложными структурами в других областях. В самом деле, можно привести множество примеров успешно функционирующих сложных систем. Некоторые из них созданы человеком, например: космический челнок Space Shuttle, туннель под Ла-Маншем, большие фирмы типа Microsoft или General Electric. В природе существуют еще более сложные системы, например система кровообращения у человека или растение.

1.2. Структура сложных систем Примеры сложных систем

Структура персонального компьютера. Персональный компьютер (ПК) — прибор умеренной сложности. Большинство ПК состоит из одних и тех же основных элементов: системной платы, монитора, клавиатуры и устройства внешней памяти какого-либо типа (гибкого или жесткого диска). Мы можем взять любую из этих частей и разложить ее в свою очередь на составляющие. Системная плата, например, содержит оперативную память, центральный процессор (ЦП) и шину, к которой подключены периферийные устройства. Каждую из этих частей можно также разложить на составляющие: ЦП состоит из регистров и схем управления, которые сами состоят из еще более простых деталей: диодов, транзисторов и т. д.

Это пример сложной иерархической системы. Персональный компьютер нормально работает благодаря четкому совместному функционированию всех его составных частей. Вместе эти части образуют логическое целое. Мы можем понять, как работает компьютер, только потому, что можем рассматривать отдельно каждую его составляющую. Таким образом, можно изучать устройства монитора и жесткого диска независимо друг от друга. Аналогично можно изучать арифметическую часть ЦП, не рассматривая при этом подсистему памяти.

Дело не только в том, что сложная система ПК иерархична, но в том, что уровни этой иерархии представляют различные уровни абстракции, причем один надстроен над другим и каждый может быть рассмотрен (понят) отдельно. На каждом уровне абстракции мы находим набор устройств, которые совместно обеспечивают некоторые функции более высокого уровня, и выбираем уровень абстракции, исходя из наших специфических потребностей. Например, пытаясь исследовать проблему синхронизации обращений к памяти, можно оставаться на уровне логических элементов компьютера, но этот уровень абстракции не подходит при поиске ошибки в прикладной программе, работающей с электронными таблицами.

Структура растений и животных. Ботаник пытается понять сходство и различия растений, изучая их морфологию, то есть форму и структуру. Растения — это сложные многоклеточные организмы. В результате совместной деятельности различных органов растений происходят такие сложные типы поведения, как фотосинтез и всасывание влаги.

Растение состоит из трех основных частей: корни, стебли и листья. Каждая из них имеет свою особую структуру. Корень, например, состоит из корневых отростков, корневых волосков, верхушки корня и т. д. Рассматривая срез листа, мы видим его эпидермис, мезофилл и сосудистую ткань. Каждая из этих структур, в свою очередь, представляет собой набор клеток. Внутри каждой

клетки можно выделить следующий уровень, который включает хлоропласт, ядро и т. д. Так же, как у компьютера, части растения образуют иерархию, каждый уровень которой обладает собственной независимой сложностью.

Все части на одном уровне абстракции взаимодействуют вполне определенным образом. Например, на высшем уровне абстракции, корни отвечают за погла-щение из почвы воды и минеральных веществ. Корни взаимодействуют со стеблями, которые передают эти вещества листьям. Листья в свою очередь используют воду и минеральные вещества, доставляемые стеблями, и производят при помощи фотосинтеза необходимые элементы.

Для каждого уровня абстракции всегда четко разграничено «внешнее» и «внутреннее». Например, можно установить, что части листа совместно обеспечивают функционирование листа в целом и очень слабо взаимодействуют или вообще прямо не взаимодействуют с элементами корней. Проще говоря, существует четкое разделение функций различных уровней абстракции.

В компьютере транзисторы используются как в схеме ЦП, так и жесткого диска. Аналогично этому большое число «унифицированных элементов» имеется во всех частях растения. Так Создатель достигал экономии средств выражения. Например, клетки служат основными строительными блоками всех структур растения; корни, стебли и листья растения состоят из клеток. И хотя любой из этих исходных элементов действительно является клеткой, существует огромное количество разнообразных клеток. Есть клетки, содержащие и не содержащие хлоропласт, клетки с оболочкой, проницаемой и непроницаемой для воды, и даже живые и умершие клетки.

При изучении морфологии растения мы не выделяем в нем отдельные части, отвечающие за отдельные фазы единого процесса, например, фотосинтеза. Фактически не существует централизованных частей, которые непосредственно координируют деятельность более низких уровней. Вместо этого мы находим отдельные части, которые действуют как независимые посредники, каждый из которых ведет себя достаточно сложно и при этом согласованно с более высокими уровнями. Только благодаря совместным действиям большого числа посредников образуется более высокий уровень функционирования растения. Наука о сложности называет это *возникающим поведением.* Поведение целого сложнее, чем поведение суммы его составляющих [б].

Обратимся к зоологии. Многоклеточные животные, как и растения, имеют иерархическую структуру: клетки формируют ткани, ткани работают вместе как органы, группы органов определяют систему (например, пищеварительную) и так далее. Мы снова вынуждены отметить присущую Создателю экономность выражения: основной строительный блок всех растений и животных — клетка. Естественно, между клетками растений и животных существуют различия. Клетки растения, например, заключены в жесткую целлюлозную оболочку в отличие от клеток животных. Но, несмотря на эти различия, обе указанные структуры, несомненно, являются клетками. Это пример общности в разных сферах.

Жизнь растений и животных поддерживает значительное число механизмов надклеточного уровня, то есть более высокого уровня абстракции. И растения, и животные используют сосудистую систему для транспортировки внутри организма питательных веществ. И у тех, и у других может существовать различие полов внутри одного вида.

Структура вещества. Исследования в таких разных областях, как астрономия и ядерная физика, дают множество других примеров невероятно сложных систем. В этих двух дисциплинах мы найдем примеры иерархических структур. Астрономы изучают галактики, которые объединены в скопления, а звезды, планеты и другие небесные тела образуют галактику. Ядерщики имеют дело со структурной иерархией физических тел совсем другого масштаба. Атомы состоят из электронов, протонов и нейтронов; электроны, по-видимому, являются элементарными частицами, но протоны, нейтроны и другие тяжелые частицы формируются из еще более мелких компонентов, называемых кварками.

Мы опять обнаруживаем общность форм механизмов в этих сложных иерархиях. На самом деле оказывается, что во Вселенной работают всего четыре типа сил: гравитационное, электромагнитное, сильное и слабое взаимодействия. Многие законы физики универсальны, например, закон сохранения энергии и импульса можно применить и к галактикам, и к кваркам.

Структура социальных институтов. Как последний пример сложных систем рассмотрим структуру общественных институтов. Люди объединяются в группы для решения задач, которые не могут быть решены индивидуально. Одни организации быстро распадаются, другие функционируют на протяжении нескольких поколений. Чем больше организация, тем отчетливее проявляется в ней иерархическая структура. Транснациональные корпорации состоят из компаний, которые в свою очередь состоят из отделений, содержащих различные филиалы. Последним принадлежат уже отдельные офисы и т. д. Границы между частями организации могут изменяться, и с течением времени может возникнуть новая, более стабильная иерархия.

Отношения между разными частями большой организации подобны отношениям между компонентами компьютера, растения или галактики. Характерно, что степень взаимодействия между сотрудниками одного учреждения несомненно выше, чем между сотрудниками двух разных учреждений. Клерк, например, обычно не общается с исполнительным директором компании, а в основном обслуживает посетителей. Но и здесь различные уровни имеют единые механизмы функционирования. Работа и клерка и директора оплачивается одной финансовой организацией, и оба они для своих целей используют общую аппаратуру, в частности, телефонную систему компании.

Пять признаков сложной системы

Исходя из такого способа изучения, можно вывести пять общих признаков любой сложной системы. Основываясь на работе Саймона и Эндо, Куртуа предлагает следующее наблюдение [7]:

1. "Сложные системы часто являются иерархическими и состоят из взаимозависимых подсистем, которые в свою очередь также могут быть разделены на подсистемы, и т. д., вплоть до самого низкого уровням."

Саймон отмечает: «тот факт, что многие сложные системы имеют почти разложимую иерархическую структуру, является главным фактором, позволяющим нам понять, описать и даже «увидеть» такие системы и их части» [8]. В самом деле, скорее всего, мы можем понять лишь те системы, которые имеют иерархическую структуру.

Важно осознать, что архитектура сложных систем складывается и из компонентов, и из иерархических отношений этих компонентов. Речтин отмечает: «Все системы имеют подсистемы, и все системы являются частями более крупных систем... Особенности системы обусловлены отношениями между ее частями, а не частями как таковыми» [9].

Что же следует считать простейшими элементами системы? Опыт подсказывает нам следующий ответ:

2. Выбор, какие компоненты в данной системе считаются элементарными, относительно произволен и в большой степени оставляется на усмотрение исследователя.

Низший уровень для одного наблюдателя может оказаться достаточно высоким для другого.

Саймон называет иерархические системы *разложимыми,* если они могут быть разделены на четко идентифицируемые части, и *почти разложимыми,* если их составляющие не являются абсолютно независимыми. Это подводит нас к следующему общему свойству всех сложных систем:

3. «Внутрикомпонентная связь обычно сильнее, чем связь между компонентами. Это обстоятельство позволяет отделять «высокочастотные» взаимодействия внутри компонентов от «низкочастотной» динамики взаимодействия между компонентами» [10].

Это различие внутрикомпонентных и межкомпонентных взаимодействий обуславливает разделение функций между частями системы и дает возможность относительно изолированно изучать каждую часть.

Как мы уже говорили, многие сложные системы организованы достаточно экономными средствами. Поэтому Саймон приводит следующий признак сложных систем:

4. «Иерархические системы обычно состоят из немногих типов подсистем, по-разному скомбинированных и организованных» [11].

Иными словам и, разные сложные системы содержат одинаковые структурные части. Эти части могут использовать общие более мелкие компоненты, такие как клетки, или более крупные структуры, типа сосудистых систем, имеющиеся и у растений, и у животных.

Выше мы отмечали, что сложные системы имеют тенденцию к развитию во времени. Саймон считает, что сложные системы будут развиваться из простых гораздо быстрее, если для них существуют устойчивые промежуточные формы [12]. Гэлл [13] выражается более эффектно:

5. «Любая работающая сложная система является результатом развития работавшей более простой системы... Сложная система, спроектированная «с нуля», никогда не заработает. Следует начинать с работающей простой системы».

В процессе развития системы объекты, первоначально рассматривавшиеся как сложные, становятся элементарными, и из них строятся более сложные системы. Более того, невозможно сразу правильно создать элементарные объекты: с ними надо сначала повозиться, чтобы больше узнать о реальном поведении системы, и затем уже совершенствовать их.

Организованная и неорганизованная сложность

Каноническая форма сложной системы. Обнаружение общих абстракций и механизмов значительно облегчает понимание сложных систем. Например, опытный пилот, сориентировавшись всего за несколько минут, может взять на себя управление многомоторным реактивным самолетом, на котором он раньше никогда не летал, и спокойно его вести. Определив элементы, общие для всех подобных самолетов (такие, как руль управления, элероны и дроссельный клапан), пилот затем найдет отличия этого конкретного самолета от других. Если пилот уже знает, как управлять одним самолетом определенного типа, ему гораздо легче научиться управлять другим похожим самолетом.

Этот пример наводит на мысль, что мы обращались с термином *иерархия* в весьма приблизительном смысле. Наиболее интересные сложные системы содержат много разных иерархий. В самолете, например, можно выделить несколько систем: питания, управления полетом и т. д. Такое разбиение дает структурную иерархию типа «быть частью». Эту же систему можно разложить совершенно другим способом. Например, турбореактивный двигатель — особый тип реактивного двигателя, а «Pratt and Whitney TF30» — особый тип турбореактивного двигателя. С другой стороны, понятие «реактивный двигатель» обобщает свойства, присущие всем реактивным двигателям; «турбореактивный двигатель» — это просто особый тип реактивного двигателя со свойствами, которые отличают его, например, от прямоточного.

Эта вторая иерархия представляет собой иерархию типа «is-a». Исходя из нашего опыта, мы сочли необходимым рассмотреть систему с двух точек зрения, как иерархию первого и второго типа. По причинам, изложенным в главе 2, мы назовем эти иерархии соответственно *структурой классов* и *структурой объектов.*[3]

Объединяя понятия структуры классов и структуры объектов с пятью признаками сложных систем, мы приходим к тому, что фактически все сложные системы можно представить одной и той же (канонической) формой, которая показана на рис. 1-1. Здесь приведены две ортогональных иерархии одной системы: классов и объектов. Каждая иерархия является многоуровневой, причем в ней классы и объекты более высокого уровня построены из более простых. Какой класс или объект выбран в качестве элементарного, зависит от рассматриваемой задачи. Объекты одного уровня имеют четко выраженные связи, особенно это касается компонентов структуры объектов. Внутри любого рассматриваемого уровня находится следующий уровень сложности. Отметим также, что структуры классов и объектов не являются независимыми: каждый элемент структуры объектов представляет специфический экземпляр определенного класса. Как видно из рис. 1-1, объектов в сложной системе обычно гораздо больше, чем классов. Показывая обе иерархии, мы демонстрируем избыточность рассматриваемой системы. Если бы мы не знали структуру классов нашей системы, нам пришлось бы повторять одни и те же сведе-

---

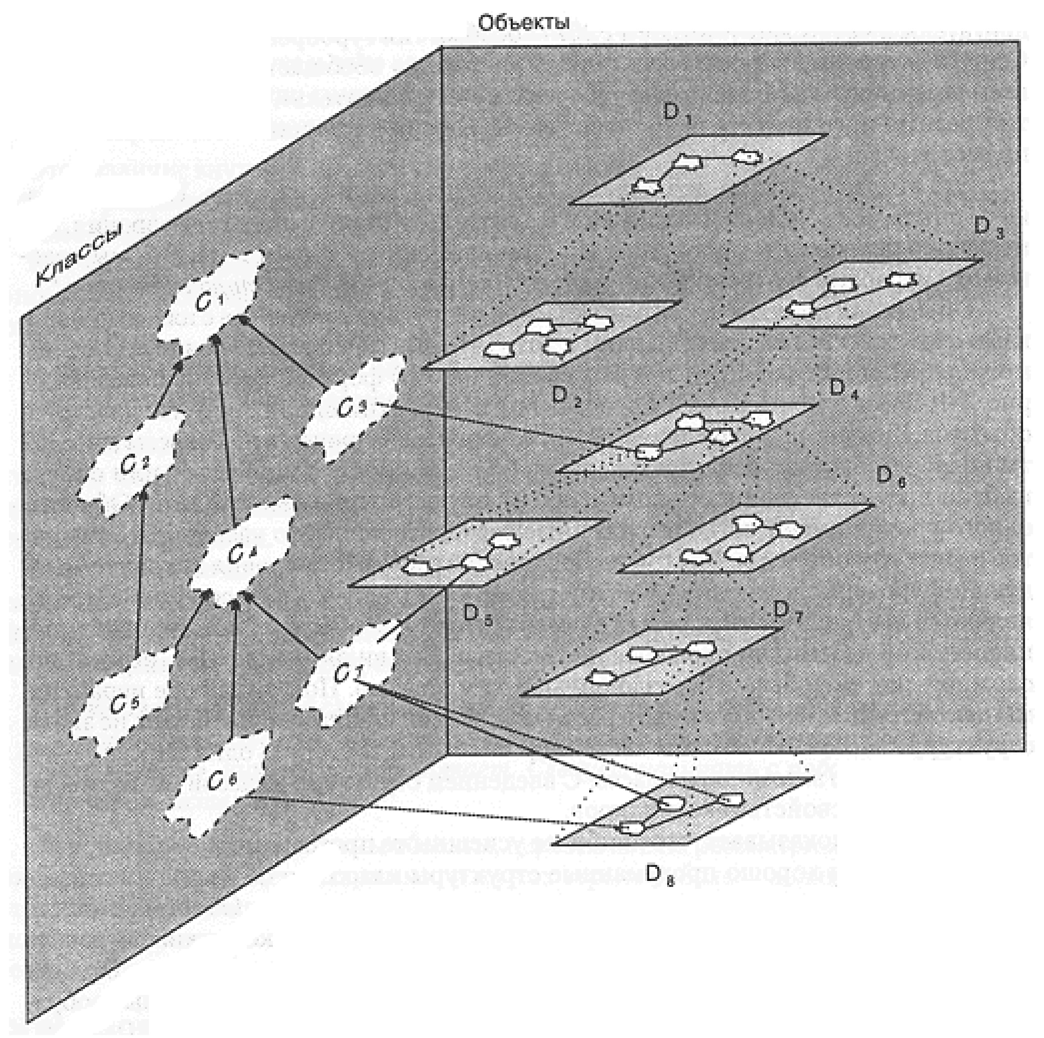
[3] Сложные программные системы включают также и другие типы иерархии. Особое значение имеют их модульная структура, которая описывает отношения между физическими компонентами системы, и иерархия процессов, которая описывает отношения между Динамическими компонентами.

ния для каждого экземпляра класса. С введением структуры классов мы размещаем в ней общие свойства экземпляров.

Наш опыт показывает, что наиболее успешны те программные системы, в которых заложены хорошо продуманные структуры классов и объектов и которые обладают пятью признаками сложных систем, описанными выше. Оценим важность этого наблюдения и выразимся более категорично: очень редко можно встретить программную систему, разработанную точно по графику, уложившуюся в бюджет и удовлетворяющую требованиям заказчика, в которой бы не были учтены соображения, изложенные выше.

Структуры классов и объектов системы вместе мы называем *архитектурой* системы.

Человеческие возможности и сложные системы. Если мы знаем, как должны быть спроектированы сложные программные системы, то почему при создании таких систем мы сталкиваемся с серьезными проблемами? Как показано в главе 2, идея о том, как бороться со сложностью программ (эту идею мы будем называть *объектный подход)* относительно нова. Существует, однако, еще одна, по-видимому, главная причина: физическая ограниченность возможностей человека при работе со сложными системами.



Когда мы начинаем анализировать сложную программную систему, в ней обнаруживается много составных частей, которые взаимодействуют друг с другом различными способами, причем ни сами части системы, ни способы их взаимодействия не обнаруживают никакого сходства. Это пример неорганизованной сложности. Когда мы начинаем организовывать систему в процессе ее проектирования, необходимо

думать сразу о многом. Например, в системе управления движением самолетов приходится одновременно контролировать состояние многих летательных аппаратов, учитывая такие их параметры, как местоположение, скорость и курс. При анализе дискретных систем необходимо рассматривать большие, сложные и не всегда детерминированные пространства состояний. К сожалению, один человек не может следить за всем этим одновременно. Эксперименты психологов, например Миллера, показывают, что максимальное количество структурных единиц информации, за которыми человеческий мозг может одновременно следить, приблизительно равно семи плюс-минус два [14]. Вероятно, это связано с объемом краткосрочной памяти у человека. Саймон также отмечает, что дополнительным ограничивающим фактором является скорость обработки мозгом поступающей информации: на восприятие каждой новой единицы информации ему требуется около 5 секунд [15].

Таким образом, мы оказались перед серьезной дилеммой. Сложность программных систем возрастает, но способность нашего мозга справиться с этой сложностью ограничена. Как же нам выйти из создававшегося затруднительного положения?

1.3. Внесение порядка в хаос

Роль декомпозиции

Как отмечает Дейкстра, «Способ управления сложными системами был известен еще в древности — *divide et impera* (разделяй и властвуй)» [16]. При проектировании сложной программной системы необходимо разделять ее на все меньшие и меньшие подсистемы, каждую из которых можно совершенствовать независимо. В этом случае мы не превысим пропускной способности человеческого мозга: для понимания любого уровня системы нам необходимо одновременно держать в уме информацию лишь о немногих ее частях (отнюдь не о всех). В самом деле, как заметил Парнас, декомпозиция вызвана сложностью программирования системы, поскольку именно эта сложность вынуждает делить пространство состояний системы [17].

Алгоритмическая декомпозиция. Большинство из нас формально обучено структурному проектированию «сверху вниз», и мы воспринимаем декомпозицию как обычное разделение алгоритмов, где каждый модуль системы выполняет один из этапов общего процесса. На рис. 1-2 приведен в качестве примера один из продуктов структурного проектирования: структурная схема, которая показывает связи между различными функциональными элементами системы. Данная структурная схема иллюстрирует часть программной схемы, изменяющей содержание управляющего файла. Она была автоматически получена из диаграммы потока данных специальной экспертной системой, которой известны правила структурного проектирования [18].

Объектно-ориентированная декомпозиция. Предположим, что у этой задачи существует альтернативный способ декомпозиции. На рис. 1-3 мы разделили систему, выбрав в качестве критерия декомпозиции принадлежность ее элементов к различным абстракциям данной проблемной области. Прежде чем разделять задачу на шаги типа *Get formatted update (Получить изменения в отформатированном виде)* и *Add check sum (Прибавить к контрольной сумме),* мы должны определить такие объекты как *Master File (Основной файл)* и *Check Sum (Контрольная сумма),* которые заимствуются из словаря предметной области.

Хотя обе схемы решают одну и ту же задачу, но они делают это разными способами. Во второй декомпозиции мир представлен совокупностью автономных действующих лиц, которые взаимодействуют друг с другом, чтобы обеспечить поведение системы, соответствующее более высокому уровню. *Get formatted update (Получить изменения в отформатированном виде)* больше не присутствует в качестве независимого алгоритма; это действие существует теперь как операция над

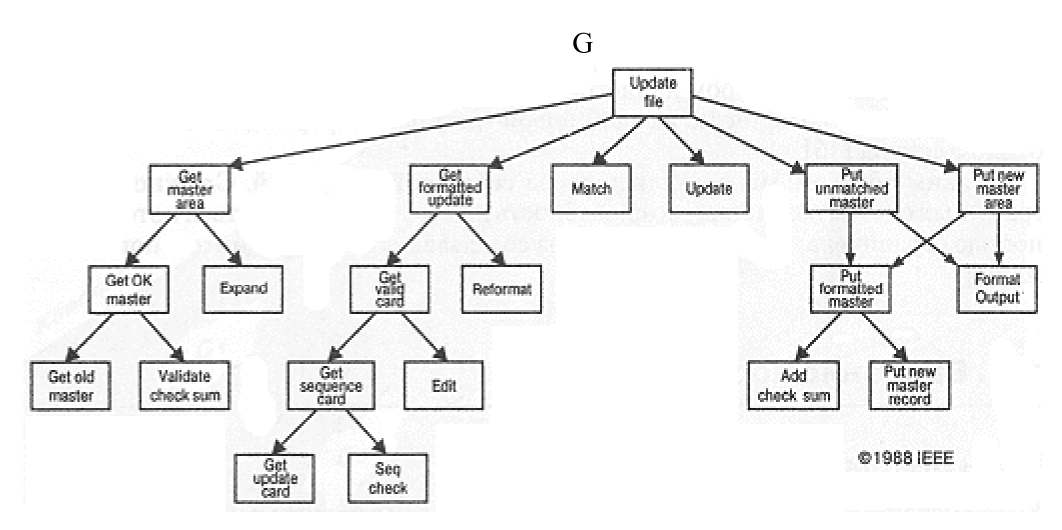


Рис. 1-2. Алгоритмическая декомпозиция

объектом *File of Updates (Файл изменений).* Эта операция создает другой объект — *Update to Card (Изменения в карте).* Таким образом, каждый объект обладает своим собственным поведением, и каждый из них моделирует некоторый объект реального мира. С этой точки зрения объект является вполне осязаемой вещью, которая демонстрирует вполне определенное поведение. Объекты что-то делают, и мы можем, послав им сообщение, попросить их выполнить то-то и то-то. Так как наша декомпозиция основана на объектах, а не на алгоритмах, мы называем ее *объектно-ориентированной декомпозицией.*

Декомпозиция: алгоритмическая или объектно-ориентированная? Какая декомпозиция сложной системы правильнее — по алгоритмам или по объектам? В этом вопросе есть подвох, и правильный ответ на него: важны оба аспекта. Разделение по алгоритмам концентрирует внимание на порядке происходящих событий, а разделение по объектам придает особое значение агентам, которые являются либо объектами, либо субъектами действия. Однако мы не можем сконструировать сложную систему одновременно двумя способами, тем более, что эти способы по сути ортогональны[4]. Мы должны начать разделение системы либо по алгоритмам, либо по объектам, а затем, используя полученную структуру, попытаться рассмотреть проблему с другой точки зрения.

Опыт показывает, что полезнее начинать с объектной декомпозиции. Такое начало поможет нам лучше справиться с приданием организованности сложности программных систем. Выше этот объектный подход помог нам при описании таких непохожих систем, как компьютеры, растения, галактики и общественные институты. Как будет видно в дальнейшем (в главах 2 и 7), объектная декомпозиция имеет несколько чрезвычайно важных преимуществ перед алгоритмической. Объект-

[4] Лэнгдон предполагает, что эта ортогональность изучалась с древних времен. Он пишет: «К. Х. Ваддингтон отметил, что такая дуальность взглядов прослеживается до древних греков. Пассивный взгляд предлагался Демокритом, который утверждал, что мир состоит из атомов. Эта позиция Демокрита ставила в центр всего материю. Классическим представителем другой стороны — активного взгляда — был Гераклит, который выделял понятие процесса»[34].

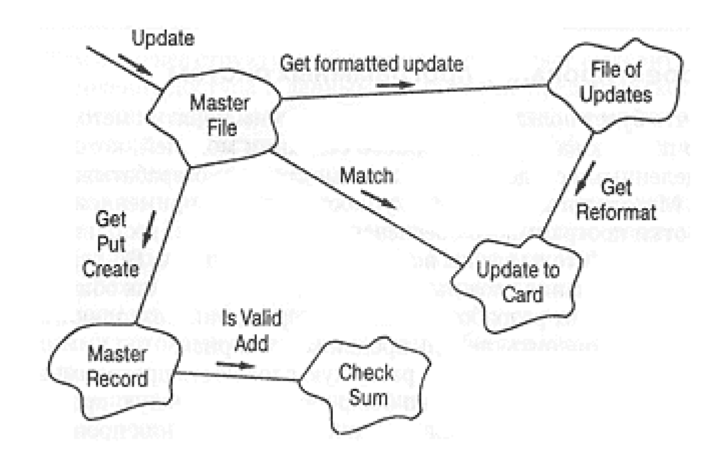


Рис. 1-3. Объектно-ориентированная декомпозиция

ная декомпозиция уменьшает размер программных систем за счет повторного использования общих механизмов, что приводит к существенной экономии выразительных средств. Объектно-ориентированные системы более гибки и проще эволюционируют со временем, потому что их схемы базируется на устойчивых промежуточных формах. Действительно, объектная декомпозиция существенно снижает риск при создании сложной программной системы, так как она развивается из меньших систем, в которых мы уже уверены. Более того, объектная декомпозиция помогает нам разобраться в сложной программной системе, предлагая нам разумные решения относительно выбора подпространства большого пространства состояний.

Преимущества объектно-ориентированных систем демонстрируются в главах 8-12 примерами прикладных программ, относящихся к различным областям. Следующая врезка сопоставляет объектно-ориентированное проектирование с более традиционными подходами.

Роль абстракции

Выше мы ссылались на эксперименты Миллера, в которых было установлено, что обычно человек может одновременно воспринять лишь 7±2 единицы информации. Это число, по-видимому, не зависит от содержания информации. Как замечает сам Миллер: «Размер нашей памяти накладывает жесткие ограничения на количество информации, которое мы можем воспринять, обработать и запомнить. Организуя поступление входной информации одновременно по нескольким различным каналам и в виде последовательности отдельных порций, мы можем прорвать... этот информационный затор» [35]. В современной терминологии это называют разбиением или выделением *абстракций.*

Вулф так описывает этот процесс: «Люди развили чрезвычайно эффективную технологию преодоления сложности. Мы абстрагируемся от нее. Будучи не в состоянии полностью воссоздать сложный объект, мы просто игнорируем не слишком важные детали и, таким образом, имеем дело с обобщенной, идеализированной моделью объекта» [36]. Например, изучая процесс фотосинтеза у растений, мы концентрируем внимание на химических реакциях в определенных клетках листа и не обращаем внимание на остальные части — черенки, жилки и т. д. И хотя мы по-прежнему вынуждены охватывать одновременно значительное количество информации, но благодаря абстракции мы пользуемся единицами информации существенно большего семантического объема. Это особенно верно, когда мы рассматриваем мир с объектно-ориентированной точки зрения, поскольку объекты как абстракции реального мира представляют собой отдельные насыщенные связные информационные единицы.

В главе 2 понятие абстракции рассмотрено более детально.

Методы проектирования программных систем

Мы решили, что будет полезно, если мы разграничим понятия метод и методология. Метод — это последовательный процесс создания моделей, которые описывают вполне определенными средствами различные стороны разрабатываемой программной системы. Методология — это совокупность методов, применяемых в жизненном цикле разработки программного обеспечения и объединенных одним общим философским подходом. Методы важны по нескольким причинам. Во-первых, они упорядочивают процесс создания сложных программных систем, как общие средства доступные для всей группы разработчиков. Во-вторых, они позволяют менеджерам в процессе разработки оценить степень продвижения и риск.

Методы появились как ответ на растущую сложность программных систем. На заре компьютерной эры очень трудно было написать большую программу, потому что возможности компьютеров были ограничены. Ограничения проистекали из объема оперативной памяти, скорости считывания информации с вторичных носителей (ими служили магнитные ленты) и быстродействия процессоров, тактовый цикл которых был равен сотням микросекунд. В 60-70-е годы эффективность применения компьютеров резко возросла, цены на них стали падать, а возможности ЭВМ увеличились. В результате стало выгодно, да и необходимо создавать все больше прикладных программ повышенной сложности. В качестве основных инструментов создания программных продуктов начали применяться алгоритмические языки высокого уровня. Эти языки расширили возможности отдельных программистов и групп разработчиков, что по иронии судьбы в свою очередь привело к увеличению уровня сложности программных систем.

В 60-70-е годы было разработано много методов, помогающих справиться с растущей сложностью программ. Наибольшее распространение получило структурное проектирование по методу сверху вниз. Метод был непосредственно основан на топологии традиционных языков высокого уровня типа FORTRAN или COBOL. В этих языках основной базовой единицей является подпрограмма, и программа в целом принимает форму дерева, в котором одни подпрограммы в процессе работы вызывают другие подпрограммы. Структурное проектирование использует именно такой подход: алгоритмическая декомпозиция применяется для разбиения большой задачи на более мелкие.

Тогда же стали появляться компьютеры еще больших, поистине гигантских возможностей. Значение структурного подхода осталось прежним, но как замечает Стейн, «оказалось, что структурный подход не работает, если объем программы превышает приблизительно 100 000 строк» [19]. В последнее время появились десятки методов, в большинстве которых устранены очевидные недостатки структурного проектирования. Наиболее удачные методы были разработаны Петерсом [20], Йеном и Цаи [21], а также фирмой Teledyne-Brown Engineering [22]. Большинство этих методов представляют собой вариации на одни и те же темы. Саммервилль предлагает разделить их на три основные группы [23]:

- метод структурного проектирования сверху вниз;
- метод потоков данных;
- объектно-ориентированное проектирование.

Примеры структурного проектирования приведены в работах Иордана и Константина [24], Майерса [25] и Пейдж-Джонса [26]. Основы его изложены в работах Вирта [27, 28], Даля, Дейкстры и Хоара [29]; интересный вариант структурного подхода можно найти в работе Милса, Лингера и Хевнера [30]. В каждом из этих подходов присутствует алгоритмическая декомпозиция. Следует отметить, что большинство существующих программ написано, по-видимому, в соответствии с одним из этих методов. Тем не менее структурный подход не позволяет выделить абстракции и обеспечить ограничение доступа к данным; он также не предоставляет достаточных средств для организации параллелизма. Структурный метод не может обеспечить создание предельно сложных систем, и он, как правило, неэффективен в объектных и объектно-ориентированных языках программирования.

Метод потоков данных лучше всего описан в ранней работе Джексона [31, 32], а также Варниера и Орра [33]. В этом методе программная система рассматривается как преобразователь

входных потоков в выходные. Метод потоков данных, как и структурный метод, с успехом применялся при решении ряда сложных задач, в частности, в системах информационного обеспечения, где существуют прямые связи между входными и выходными потоками системы и где не требуется уделять особого внимания быстродействию.

Объектно-ориентированное проектирование (object-oriented design, OOD) — это подход, основы которого изложены в данной книге. В основе OOD лежит представление о том, что программную систему необходимо проектировать как совокупность взаимодействующих друг с другом объектов, рассматривая каждый объект как экземпляр определенного класса, причем классы образуют иерархию. Объектно-ориентированный подход отражает топологию новейших языков высокого уровня, таких как Smalltalk, Object Pascal, C++, CLOS и Ada.

Роль иерархии

Другим способом, расширяющим информационные единицы, является организация внутри системы иерархий классов и объектов. Объектная структура важна, так как она иллюстрирует схему взаимодействия объектов друг с другом, которое осуществляется с помощью *механизмов* взаимодействия. Структура классов не менее важна: она определяет общность структур и поведения внутри системы. Зачем, например, изучать фотосинтез каждой клетки отдельного листа растения, когда достаточно изучить одну такую клетку, поскольку мы ожидаем, что все остальные ведут себя подобным же образом. И хотя мы рассматриваем каждый объект определенного типа как отдельный, можно предположить, что его поведение будет похоже на поведение других объектов того же типа. Классифицируя объекты по группам родственных абстракций (например, типы клеток растений в противовес клеткам животных), мы четко разделяем общие и уникальные свойства разных объектов, что помогает нам затем справляться со свойственной им сложностью [37].

Определить иерархии в сложной программной системе не всегда легко, так как это требует разработки моделей многих объектов, поведение каждого из которых может отличаться чрезвычайной сложностью. Однако после их определения, структура сложной системы и, в свою очередь, наше понимание ее сразу во многом проясняются. В главе 3 детально рассматривается природа иерархий классов и объектов, а в главе 4 описываются приемы распознавания этих структур.

1.4. О проектировании сложных систем

Инженерное дело как наука и искусство

На практике любая инженерная дисциплина, будь то строительство, механика, химия, электроника или программирование, содержит в себе элементы и науки, и искусства. Петроски красноречиво утверждает: «Разработка новых структур предполагает и полет фантазии, и синтез опыта и знаний: все то, что необходимо художнику для реализации своего замысла на холсте или бумаге. После того, как этот замысел созрел в голове инженера-художника, он обязательно должен быть проанализирован с точки зрения применимости данного научного метода инженером-ученым со всей тщательностью, присущей настоящему ученому» [38]. Аналогично, Дейкстра отмечает, что «Программная постановка задачи является упражнением в применении абстракции и требует способностей как формального математика, так и компетентного инженера» [39].

Когда разрабатывается совершенно новая система, роль инженера как художника выдвигается на первый план. Это происходит постоянно при проектировании программ. А тем более при работе с системами, обладающими обратной связью, и особенно в случае систем управления и контроля, когда нам приходится писать программное обеспечение, требования к которому нестандартны, и к тому же для специально сконструированного процессора. В других случаях, например, при создании прикладных научных средств, инструментов для исследований в области искусственного интеллекта или даже для систем обработки информации, требования к системе могут быть хорошо и точно определены, но определены таким образом, что

соответствующий им технический уровень разработки выходит за пределы существующих технологий. Нам, например, могут предложить создать систему, обладающую большим быстродействием, большей вместимостью или имеющей гораздо более мощные функциональные возможности по сравнению с уже существующими. Во всех этих случаях мы будем стараться использовать знакомые абстракции и механизмы («устойчивые промежуточные формы» в терминах Саймона) как основу новой системы. При наличии большой библиотеки повторно используемых программных компонентов, инженер-программист должен их по-новому скомпоновать, чтобы удовлетворить всем явным и неявным требованиям к системе, точно так же, как художник или музыкант находит новые возможности своего инструмента. Но так как подобных богатых библиотек практически не существует, инженер-программист обычно может использовать, к сожалению, лишь относительно небольшой список готовых модулей.

Смысл проектирования

В любой инженерной дисциплине под проектированием обычно понимается некий унифицированный подход, с помощью которого мы ищем пути решения определенной проблемы, обеспечивая выполнение поставленной задачи. В контексте инженерного проектирования Мостов определил цель проектирования как создание системы, которая

- «удовлетворяет заданным (возможно, неформальным) функциональным спецификациям;
- согласована с ограничениями, накладываемыми оборудованием;
- удовлетворяет явным и неявным требованиям по эксплуатационным качествам и ресурсопотреблению;
- удовлетворяет явным и неявным критериям дизайна продукта;
- удовлетворяет требованиям к самому процессу разработки, таким, например, как продолжительность и стоимость, а также привлечение дополнительных инструментальных средств» [40].

По предположению Страуструпа: «Цель проектирования — выявление ясной и относительно простой внутренней структуры, иногда называемой архитектурой... Проект есть окончательный продукт процесса проектирования» [41]. Проектирование подразумевает учет противоречивых требований. Его продуктами являются модели, позволяющие нам понять структуру будущей системы, сбалансировать требования и наметить схему реализации.

Важность построения модели. Моделирование широко распространено во всех инженерных дисциплинах, в значительной степени из-за того, что оно реализует принципы декомпозиции, абстракции и иерархии [42]. Каждая модель описывает определенную часть рассматриваемой системы, а мы в свою очередь строим новые модели на базе старых, в которых более или менее уверены. Модели позволяют нам контролировать наши неудачи. Мы оцениваем поведение каждой модели в обычных и необычных ситуациях, а затем проводим соответствующие доработки, если нас что-то не удовлетворяет.

Как мы уже сказали выше, чтобы понять во всех тонкостях поведение сложной системы, приходится использовать не одну модель. Например, проектируя компьютер на одной плате, инженер-электронщик должен рассматривать систему как на уровне отдельных элементов схемы (микросхем), так и на уровне схемы. Схема помогает инженеру разобраться в совместном поведении микросхем. Схема представляет собой план физической реализации системы микросхем, в котором учтены размер платы, потребляемая мощность и типы имеющихся интегральных микросхем. С этой точки зрения инженер может независимо оценивать такие параметры системы, как температурное распределение и технологичность изготовления. Проектировщик платы может также рассматривать динамические и статические особенности системы. Аналогично, инженер-электронщик использует диаграммы, иллюстрирующие статические связи между различными микросхемами, и временные диаграммы, отражающие

поведение элементов во времени. Затем инженер может применить осциллограф или цифровой анализатор для проверки правильности и статической, и динамической моделей.

Элементы программного проектирования. Ясно, что не существует такого универсального метода, «серебряной пули» [43], который бы провел инженера-программиста по пути от требований к сложной программной системе до их выполнения. Проектирование сложной программной системы отнюдь не сводится к слепому следованию некоему набору рецептов. Скорее это постепенный и итеративный процесс. И тем не менее использование методологии проектирования вносит в яроцесс разработки определенную организованность. Инженеры-программисты разработали десятки различных методов, которые мы можем классифицировать

по трем категориям. Несмотря на различия, эти методы имеют что-то общее. Их, в частности, объединяет следующее:

- условные обозначения — язык для описания каждой модели;
- процесс — правила проектирования модели;
- инструменты — средства, которые ускоряют процесс создания моделей, и в которых уже воплощены законы функционирования моделей. Инструменты помогают выявлять ошибки в процессе разработки.

Хороший метод проектирования базируется на прочной теоретической основе и при этом дает программисту известную степень свободы самовыражения.

Объектно-ориентированные модели. Существует ли наилучший метод проектирования? На этот вопрос нет однозначного ответа. По сути дела это завуалированный предыдущий вопрос: "Существует ли лучший способ декомпозиции сложной системы?" Если и существует, то пока он никому не известен. Этот вопрос можно поставить следующим образом: «Как наилучшим способом разделить сложную систему на подсистемы?» Еще раз напомним, что полезнее всего создавать такие модели, которые фокусируют внимание на объектах, найденных в самой предметной области, и образуют то, что мы назвали *объектно-ориентированной декомпозицией.*

Объектно-ориентированный анализ и проектирование — это метод, логически приводящий нас к объектно-ориентированной декомпозиции. Применяя объектно-ориентированное проектирование, мы создаем гибкие программы, написанные экономными средствами. При разумном разделении пространства состояний мы добиваемся большей уверенности в правильности нашей программы. В итоге, мы уменьшаем риск при разработке сложных программных систем.

Так как построение моделей крайне важно при проектировании сложных систем, объектно-ориентированное проектирование предлагает богатый выбор моделей, которые представлены на рис. 1-4. Объектно-ориентированные модели проектирования отражают иерархию и классов, и объектов системы. Эти модели покрывают весь спектр важнейших конструкторских решений, которые необходимо рассматривать при разработке сложной системы, и таким образом вдохновляют нас на создание проектов, обладающих всеми пятью атрибутами хорошо организованных сложных систем.

В главе 5 подробно рассмотрен каждый из четырех типов моделей. В главе 6 описан процесс объектно-ориентированного проектирования, представляющий собой цепь последовательных шагов по созданию и развитию моделей. В главе 7 рассмотрена практика управления процессом объектно-ориентированного проектирования.

В этой главе мы привели доводы в пользу применения объектно-ориентированного анализа и проектирования для преодоления сложности, связанной с разработкой программных систем. Кроме того, мы определили ряд фундаментальных преимуществ, достигаемых в результате применения такого подхода. Прежде чем мы представим систему обозначений и процесс проектирования, мы должны изучить принципы, на которых этот процесс проектирования основан: абстрагирование, инкапсуляцию, модульность, иерархию, типизацию, параллелизм и устойчивость.

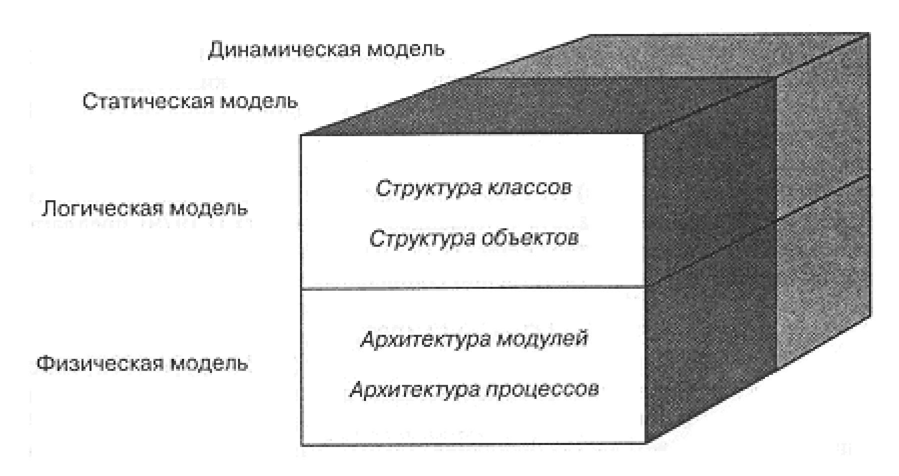


Рис. 1-4. Объектно-ориентированные модели

Выводы

- Программам присуща сложность, которая нередко превосходит возможности человеческого разума.
- Задача разработчиков программных систем — создать у пользователя разрабатываемой системы иллюзию простоты.
- Сложные структуры часто принимают форму иерархий; полезны обе иерархии: и классов, и объектов.
- Сложные системы обычно создаются на основе устойчивых промежуточных форм.
- Познавательные способности человека ограничены; мы можем раздвинуть их рамки, используя декомпозицию, выделение абстракций и создание иерархий.
- Сложные системы можно исследовать, концентрируя основное внимание либо на объектах, либо на процессах; имеются веские основания использовать объектно-ориентированную декомпозицию, при которой мир рассматривается как упорядоченная совокупность объектов, которые в процессе взаимодействия друг с другом определяют поведение системы.
- Объектно-ориентированный анализ и проектирование — метод, использующий объектную декомпозицию; объектно-ориентированный подход имеет свою систему условных обозначений и предлагает богатый набор логических и физических моделей, с помощью которых мы можем получить представление о различных аспектах рассматриваемой системы.

Дополнительная литература

Проблемы, связанные с развитием сложных программных систем, были отчетливо описаны в классических работах Брукса (Brooks) [H 1975] и [H 1987]. В работах Гласса (Glass) [H 1982], Defense Science Board [H 1987], и Joint Service Task Force [H 1982] можно найти более свежую информацию о современной практике программирования. Эмпирические исследования природы и причин программистских неудач можно найти в работах ван Генучтена (van Genuchten) [H 1991], Гвиндона (Guindon) и др. [H 1987], Джонса (Jones) [H1992].

Работы Саймона (Simon) [A 1962,1982] — богатый источник сведений об архитектуре сложных систем. Куртуа (Courtois) [A 1985] применил эти идеи к области программного обеспечения. Плодотворная работа Александера (Alexander) [11979] предлагает свежий подход к

архитектуре. Питер (Peter) [I 1986] и Петроски (Petroski) [11985] изучали сложность в контексте соответственно социальных и физических систем. Аллен и Стар (Alien and Starr) [A 1982] изучали иерархические системы в ряде предметных областей. Флуд и Кэрсон (Flood and Carson) [A 1988] предприняли формальное исследование сложности сквозь призму теории систем. Волдрап (Waldrop) [A 1992] описал возникающую науку о сложности и ее использование при изучении больших адаптивных систем, возникающего поведения и самоорганизации. Отчет Миллера (Miller) [A 1956] дает эмпирические свидетельства фундаментальных ограничивающих факторов человеческого сознания.

По проектированию программного обеспечения есть ряд замечательных ссылок. Росс, Гудинаф и Ирвайн (Ross, Goodenough, and Irvine) [H 1980], а также Зелковитс (Zeikowitz) [H 1978] — это две классические работы, суммирующие существенные элементы проектирования. Более широкий круг работ по этому предмету включает: Дженсен и Тонис (Jensen and Tonies) [H 1979], Саммервиль (Sommerville) [H 1985], Вик и Рамамурти (Vick and Ramamourthy) [H 1984], Вегнер (Wegner) [H 1980], Пресман (Pressman) [H 1992], Оман и Льюис (Oman and Lewis) [A 1990], Берзинс и Луки (Berzins and Luqi) [H 1991] и Нг и Йен(NgandYen) [H 1990]. Другие статьи, касающиеся проектирования программного обеспечения , можно найти в Йордон (Yourdon) [H 1979] и Фриман и Вассерман(Freeman and Wasserman)[H 1993]. Две работы, Грэхема (Graham) [F 1991] и Берарда (Berard) [H 1993], предлагают широкое истолкование объектно-ориентированного проектирования.

Глейк (Gleik) [I 1987] предложил легко читаемое введение в хаосоведение.

Глава 2

Объектная модель

Объектно-ориентированная технология основывается на так называемой *объектной модели.* Основными ее принципами являются: абстрагирование, инкапсуляция, модульность, иерархичность, типизация, параллелизм и сохраняемость. Каждый из этих принципов сам по себе не нов, но в объектной модели они впервые применены в совокупности.

Объектно-ориентированный анализ и проектирование принципиально отличаются от традиционных подходов структурного проектирования: здесь нужно по-другому представлять себе процесс декомпозиции, а архитектура получающегося программного продукта в значительной степени выходит за рамки представлений, традиционных для структурного программирования. Отличия обусловлены тем, что структурное проектирование основано на структурном программировании, тогда как в основе объектно-ориентированного проектирования лежит методология объектно-ориентированного программирования, К сожалению, для разных людей термин «объектно-ориентированное программирование» означает разное. Ренч правильно предсказал: «В 1980-х годах объектно-ориентированное программирование будет занимать такое же место, какое занимало структурное программирование в 1970-х. но всем будет нравиться. Каждая фирма будет рекламировать свой продукт как зданный по этой технологии. Все программисты будут писать в этом стиле, причем все по-разному. Все менеджеры будут рассуждать о нем. И никто не будет знать, что же это такое[5]» [1]. Данные предсказания продолжают сбываться и в 1990-х годах.

В этой главе мы выясним, чем является и чем не является объектно-ориентированная разработка программ, и в чем отличия этого подхода к проектированию от других с учетом семи перечисленных выше элементов объектной модели.

2.1. Эволюция объектной модели

Тенденции в проектировании

Поколения языков программирования. Оглядываясь на короткую, но колоритную историю развития программирования, нельзя не заметить две сменяющих друг друга тенденции:

- смещение акцентов от программирования отдельных деталей к программированию более крупных компонент;
- развитие и совершенствование языков программирования высокого уровня.

Большинство современных коммерческих программных систем больше и существенно сложнее, чем были их предшественники даже несколько лет тому назад. Этот рост сложности вызвал большое число прикладных исследований по методологии проектирования, особенно, по декомпозиции, абстрагированию и иерархиям. Создание более выразительных языков программирования пополнило достижения в этой области. Возникла тенденция перехода от языков, указывающих компьютеру, что делать (императивные языки), к языкам, описывающим ключевые абстракции проблемной области (декларативные языки).

Вегнер сгруппировал некоторые из наиболее известных языков высокого уровня в четыре поколения в зависимости от того, какие языковые конструкции впервые в них появились:

- Первое поколение(1954-1958)

FORTRAN I	Математические формулы
ALGOL-58	Математические формулы
Flowmatic	Математические формулы
IPL V	Математические формулы

- Второе поколение(1959-1961)

FORTRAN II	Подпрограммы, раздельная компиляция
ALGOL-60	Блочная структура, типы данных
COBOL	Описание данных, работа с файлами
Lisp	Обработка списков, указатели, сборка мусора

[5] Wegner, P. [J 1981].

- Третье поколение(1962-1970)

PL/I	FORTRAN+ALGOL+COBOL
ALGOL-68	Более строгий приемник ALGOL-60
Pascal	Более простой приемник ALGOL-60
Simula	Классы, абстрактные данные

- Потерянное поколение (1970-1980)

Много языков созданных, но мало выживших[6] [2].

В каждом следующем поколении менялись поддерживаемые языками механизмы абстракции. Языки первого поколения ориентировались на научно-инженерные применения, и словарь этой предметной области был почти исключительно математическим. Такие языки, как FORTRAN I, были созданы для упрощения программирования математических формул, чтобы освободить программиста от трудностей ассемблера и машинного кода. Первое поколение языков высокого уровня было шагом, приближающим программирование к предметной области и удаляющим от конкретной машины. Во втором поколении языков основной тенденцией стало развитие алгоритмических абстракций. В это время мощность компьютеров быстро росла, а компьютерная индустрия позволила расширить области их применения, особенно в бизнесе. Главной задачей стало инструктировать машину, что делать: сначала прочти эти анкеты сотрудников, затем отсортируй их и выведи результаты на печать. Это было еще одним шагом к предметной области и от конкретной машины. В конце 60-х годов с появлением транзисторов, а затем интегральных схем, стоимость компьютеров резко снизилась, а их производительность росла почти экспоненциально. Появилась возможность решать все более сложные задачи, но это требовало умения обрабатывать самые разнообразные типы данных. Такие языки как ALGOL-68 и затем Pascal стали поддерживать абстракцию данных. Программисты смогли описывать свои собственные типы данных. Это стало еще одним шагом к предметной области и от привязки к конкретной машине.

70-е годы знаменовались безумным всплеском активности: было создано около двух тысяч различных языков и их диалектов. Неадекватность более ранних языков написанию крупных программных систем стала очевидной, поэтому новые языки имели механизмы, устраняющие это ограничение. Лишь немногие из этих языков смогли выжить (попробуйте найти свежий учебник по языкам Fred, Chaos, Tranquil), однако многие их принципы нашли отражение в новых версиях более ранних языков. Таким образом, мы получили языки Smalltalk (новаторски переработанное наследие Simula), Ada (наследник ALGOL-68 и Pascal с элементами Simula, Alphard и CLU), CLOS (объединивший Lisp, LOOPS и Flavors), C++ (возникший от брака C и Simula) и Eiffel (произошел от Simula и Ada). Наибольший интерес для дальнейшего изложения представляет класс языков, называемых *объектными* и *объектно-ориентированными,* которые в наибольшей степени отвечают задаче объектно-ориентированной декомпозиции программного обеспечения.

Топология языков первого и начала второго поколения. Для пояснения сказанного рассмотрим структуры, характерные для каждого поколения. На рис. 2-1 показана топология, типичная для большинства языков первого поколения и первой стадии второго поколения. Говоря «топология», мы имеем в виду основные элементы языка программирования и их взаимодействие. Можно отметить, что для таких языков, как FORTRAN и COBOL, основным строительным блоком является подпрограмма (параграф в терминах COBOL). Программы, реализованные на таких языках, имеют относительно простую структуру, состоящую только из глобальных данных и подпрограмм. Стрелками на рисунке обозначено влияние подпрограмм на данные. В процессе разработки можно логически разделить разнотипные Данные, но механизмы языков практически не поддерживают такого разделения. Ошибка в какой-либо части программы может иметь далеко идущие последствия, так как область данных открыта всем подпрограммам. В больших системах трудно гарантировать целостность данных при внесении изменений в какую-либо часть системы. В процессе эксплуатации уже через короткое время возникает путаница

---

[6] Последняя фраза, очевидно, следует евангельскому «...много званных, но мало избранных» (Матф. 22:14). — *Примеч. ред.*

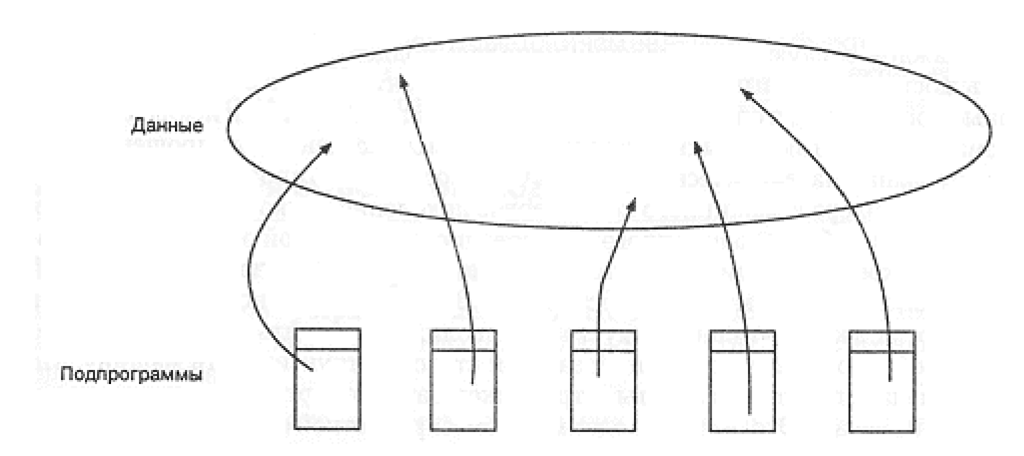


Рис. 2-1. Топология языков первого и начала второго поколения

из-за большого количества перекрестных связей между подпрограммами, запутанных схем управления, неясного смысла данных, что угрожает надежности системы и определенно снижает ясность программы.

Топология языков позднего второго и раннего третьего поколения. Начиная с середины 60-х годов стали осознавать роль подпрограмм как важного промежуточного звена между решаемой задачей и компьютером [3]. Шоу отмечает: «Первая программная абстракция, названная процедурной абстракцией, прямо вытекает из этого прагматического взгляда на программные средства... Подпрограммы возникли до 1950 года, но тогда они не были оценены в качестве абстракции... Их рассматривали как средства, упрощающие работу... Но очень скоро стало ясно, что подпрограммы это абстрактные программные функции» [4].

Использование подпрограмм как механизма абстрагирования имело три существенных последствия. Во-первых, были разработаны языки, поддерживавшие разнообразные механизмы передачи параметров. Во-вторых, были заложены основания структурного программирования, что выразилось в языковой поддержке механизмов вложенности подпрограмм и в научном исследовании структур управления и областей видимости. В-третьих, возникли методы структурного проектирования, стимулирующие разработчиков создавать большие системы, используя подпрограммы как готовые строительные блоки. Архитектура языков программирования этого периода (рис. 2-2), как и следовало ожидать, представляет собой вариации на темы предыдущего поколения. В нее внесены кое-какие усовершенствования, в частности, усилено управление алгоритмическими абстракциями, но остается нерешенной проблема программирования «в большом» и проектирования данных.

Топология языков конца третьего поколения. Начиная с FORTRAN II и далее, для решения задач программирования «в большом» начал развиваться новый важный механизм структурирования. Разрастание программных проектов означало увеличение размеров и коллективов программистов, а, следовательно, необходимость независимой разработки отдельных частей проекта. Ответом на эту потребность стал отдельно компилируемый модуль, который сначала был просто более или менее случайным набором данных и подпрограмм (рис. 2-3). В такие модули собирали подпрограммы, которые, как казалось, скорее всего будут изменяться со

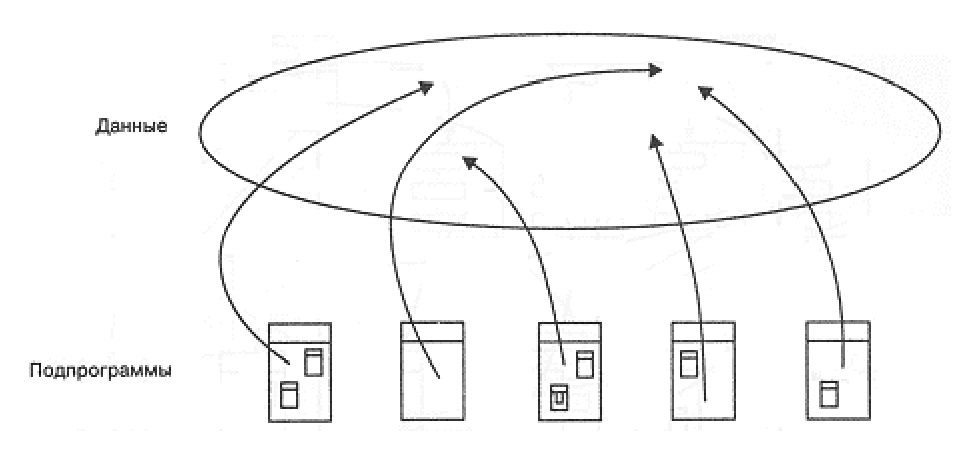


Рис. 2-2. Топология языков позднего второго и раннего третьего поколения

вместно, и мало кто рассматривал их как новую технику абстракции. В большинстве языков этого поколения, хотя и поддерживалось модульное программирование, но не вводилось никаких правил, обеспечивающих согласование интерфейсов модулей. Программист, сочиняющий подпрограмму в одном из модулей, мог, например, ожидать, что ее будут вызывать с тремя параметрами: действительным числом, массивом из десяти элементов и целым числом, обозначающим логическое значение. Но в каком-то другом модуле, вопреки предположениям автора, эта подпрограмма могла по ошибке вызываться с фактическими параметрами в виде: целого числа, массива из пяти элементов и отрицательного числа. Аналогично, один из модулей мог завести общую область данных и считать, что это его собственная область, а другой модуль мог нарушить это предположение, свободно манипулируя с этими данными. К сожалению, поскольку большинство языков предоставляло в лучшем случае рудиментарную поддержку абстрактных данных и типов, такие ошибки выявлялись только при выполнении программы.

Топология объектных и объектно-ориентированных языков. Значение абстрактных типов данных в разрешении проблемы сложности систем хорошо выразил Шан-кар: «Абстрагирование, достигаемое посредством использования процедур, хорошо подходит для описания абстрактных действий, но не годится для описания абстрактных объектов. Это серьезный недостаток, так как во многих практических ситуациях сложность объектов, с которыми нужно работать, составляет основную часть сложности всей задачи» [5]. Осознание этого влечет два важных вывода. Во-первых, возникают методы проектирования на основе потоков данных, которые вносят упорядоченность в абстракцию данных в языках, ориентированных на алгоритмы. Во-вторых, появляется теория типов, которая воплощается в таких языках, как Pascal.

Естественным завершением реализации этих идей, начавшейся с языка Simula и развитой в последующих языках в 1970-1980-е годы, стало сравнительно недавнее появление таких языков, как Smalltalk, Object Pascal, C++, CLOS, Ada и Eiffel. По причинам, которые мы вскоре объясним, эти языки получили название *объектных* или *объектно-ориентированных.* На рис. 2-4 приведена топология таких языков применительно к задачам малой и средней степени сложности. Основным элементом конструкции в указанных языках служит *модуль,* составленный из логически связанных классов и объектов, а не подпрограмма, как в языках первого поколения.

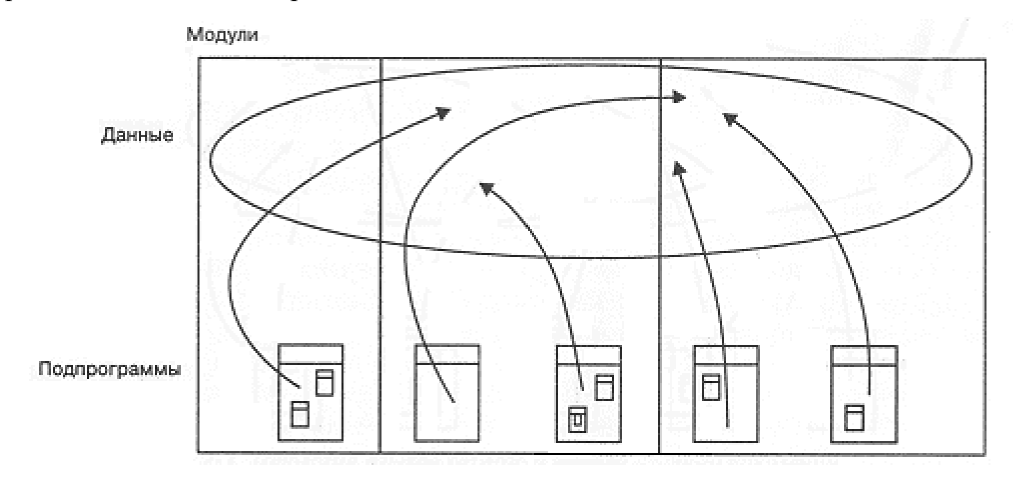


Рис. 2-3. Топология языков конца третьего поколения

Другими словами: «Если процедуры и функции — глаголы, а данные — существительные, то процедурные программы строятся из глаголов, а объектно-ориентированные — из существительных» [б]. По этой же причине структура программ малой и средней сложности при объектно-ориентированном подходе представляется графом, а не деревом, как в случае алгоритмических языков. Кроме того, уменьшена или отсутствует область глобальных данных. Данные и действия организуются теперь таким образом, что основными логическими строительными блоками наших систем становятся классы и объекты, а не алгоритмы.

В настоящее время мы продвинулись много дальше программирования «в большом» и предстали перед программированием «в огромном». Для очень сложных систем классы, объекты и модули являются необходимыми, но не достаточными средствами абстракции. К счастью, объектный подход масштабируется и может быть применен на все более высоких уровнях. Кластеры абстракций в больших системах могут представляться в виде многослойной структуры. На каждом уровне можно выделить

группы объектов, тесно взаимодействующих для решения задачи более высокого уровня абстракции. Внутри каждого кластера мы неизбежно найдем такое же множество взаимодействующих абстракций (рис. 2-5). Это соответствует подходу к сложным системам, изложенному в главе 1.

Основные положения объектной модели

Методы структурного проектирования помогают упростить процесс разработки сложных систем за счет использования алгоритмов как готовых строительных блоков. Аналогично, методы объектно-ориентированного проектирования созданы, чтобы помочь разработчикам применять мощные выразительные средства объектного и объектно-ориентированного программирования, использующего в качестве блоков классы и объекты.

Но в объектной модели отражается и множество других факторов. Как показано во врезке ниже, объектный подход зарекомендовал себя как унифицирующая идея всей компьютерной науки, применимая не только в программировании, но также в проектировании интерфейса пользователя, баз данных и даже архитекту-

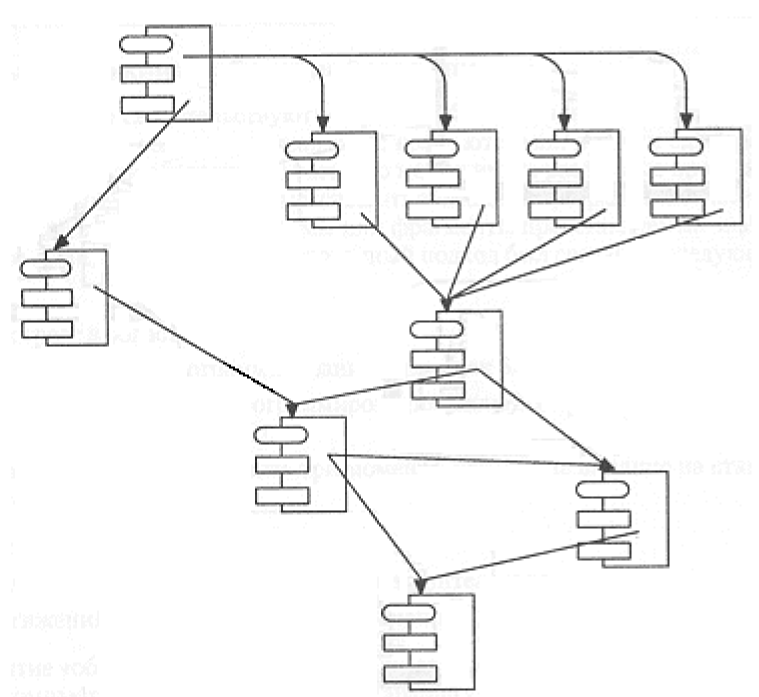


Рис. 2-4. Топология малых и средних приложений в объектных и объектно-ориентированных языках

ры компьютеров. Причина такой широты в том, что ориентация на объекты позволяет нам справляться со сложностью систем самой разной природы.

Объектно-ориентированный анализ и проектирование отражают эволюционное, а не революционное развитие проектирования; новая методология не порывает с прежними методами, а строится с учетом предшествующего опыта. К сожалению, большинство программистов в настоящее время формально и неформально натренированы на применение только методов структурного проектирования. Разумеется, многие хорошие проектировщики создали и продолжают совершенствовать большое количество программных систем на основе этой методологии. Однако алгоритмическая декомпозиция помогает только до определенного предела, и обращение к объектно-ориентированной декомпозиции необходимо. Более того, при попытках использовать такие языки, как C++ или Ada, в качестве традиционных, алгоритмически ориентированных, мы не только теряем их внутренний потенциал — скорее всего результат будет даже хуже, чем при использовании обычных языков C и Pascal. Дать электродрель плотнику, который не слышал об электричестве, значит использовать ее в качестве молотка. Он согнет несколько гвоздей и разобьет себе пальцы, потому что электродрель мало пригодна для замены молотка.

OOP, OOD и OOA

Унаследовав от многих предшественников, объектный подход, к сожалению, перенял и запутанную терминологию. Программист в Smalltalk пользуется термином *метод,* в C++ — термином *виртуальная функция,* в CLOS — *обобщенная функция.*

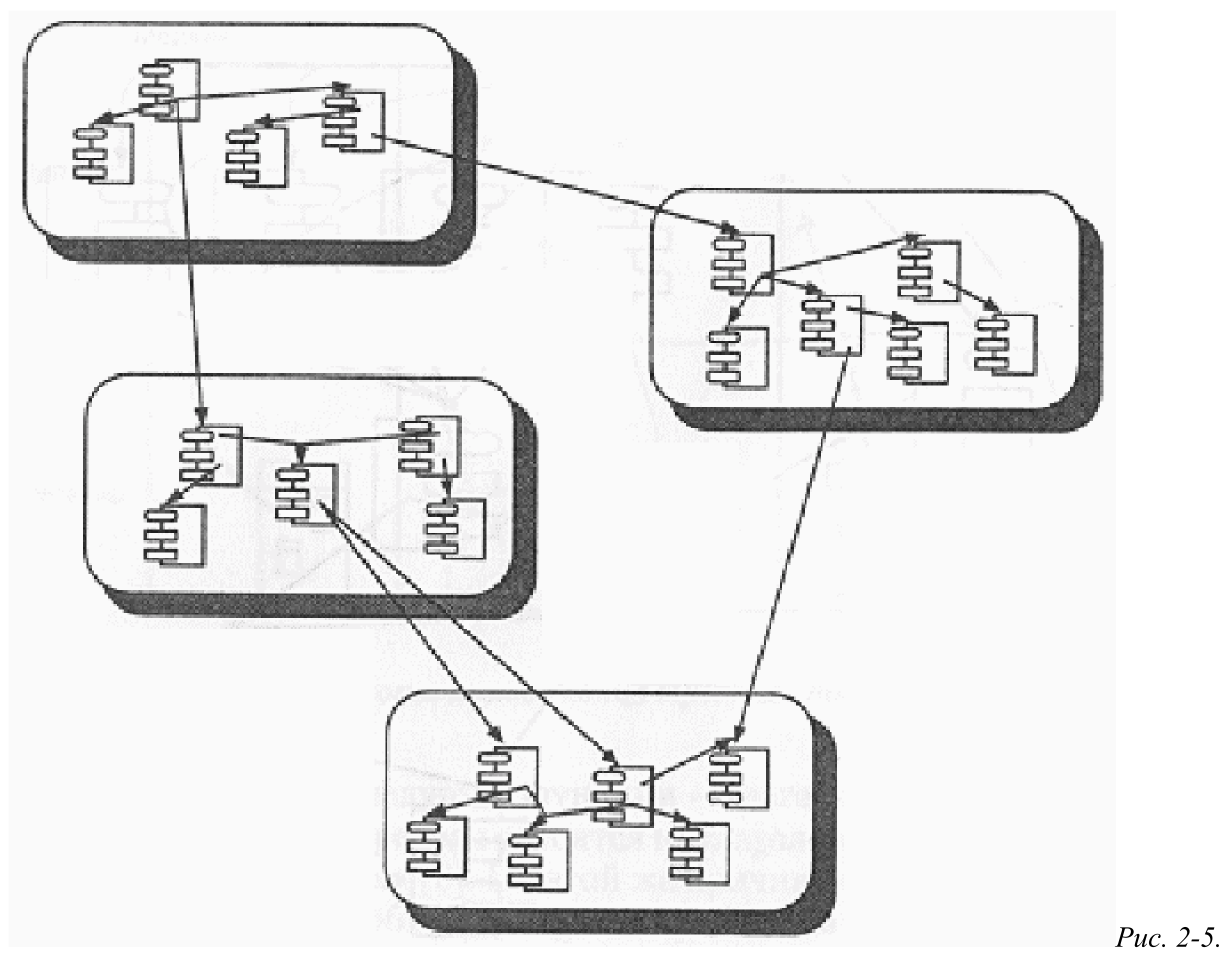


Рис. 2-5. Топология больших приложений в объектных и объектно-ориентированных языках

В Object Pascal используется термин *приведение типов,* а в языке Ada то же самое называется *преобразование типов.* Чтобы уменьшить путаницу, следует определить, что является объектно-ориентированным, а что — нет. Определение наиболее употребительных терминов и понятий вы найдете в глоссарии в конце книги.

Термин *объектно-ориентированный,* по мнению Бхаскара, «затаскан до потери смысла, как "материнство", "яблочный пирог" и "структурное программирование"» [7]. Можно согласиться, что понятие объекта является центральным во всем, что относится к объектно-ориентированной методологии. В главе 1 мы определили объект как осязаемую сущность, которая четко проявляет свое поведение. Стефик и Бобров определяют объекты как «сущности, объединяющие процедуры и данные, так как они производят вычисления и сохраняют свое локальное состояние» [8]. Определение объекта как сущности в какой-то мере отвечает на вопрос, но все же главным в понятии объекта является объединение идей абстракции данных и алгоритмов. Джонс уточняет это понятие следующим образом: «В объектном подходе акцент переносится на конкретные характеристики физической или абстрактной системы, являющейся предметом программного моделирования... Объекты обладают целостностью, которая не должна — а, в действительности, не может — быть нарушена. Объект может только менять состояние, вести себя, управляться или становиться в определенное отношение к другим объектам. Иначе говоря, свойства, которые характеризуют объект и его поведение, остаются неизменными. Например, лифт характеризуется теми неизменными свойствами, что он может двигаться вверх и вниз, оставаясь в пределах шахты... Любая модель должна учитывать эти свойства лифта, так как они входят в его определение» [32].

Основные положения объектной модели

Йонесава и Токоро свидетельствуют: «термин "объект" появился практически независимо в различных областях, связанных с компьютерами, и почти одновременно в начале 70-х годов для обозначения того, что может иметь различные проявления, оставаясь целостным. Для того, чтобы уменьшить сложность программных систем, объектами назывались компоненты системы или фрагменты представляемых знании» [9]. По мнению Леви, объектно-ориентированный подход был связан со следующими событиями:

- «прогресс в области архитектуры ЭВМ
- развитие языков программирования, таких как Simula, Smalltalk, CLU, Ada
- развитие методологии программирования, включая принципы модульности и скрытия данных» [10].

К этому еще следует добавить три момента, оказавшие влияние на становление объектного подхода:

- развитие теории баз данных
- исследования в области искусственного интеллекта
- достижения философии и теории познания.

Понятие «объект» впервые было использовано более 20 лет назад при конструировании компьютеров с descriptor-based и capability-based архитектурами [11]. В этих работах делались попытки отойти от традиционной архитектуры фон Неймана и преодолеть барьер между высоким уровнем программной абстракции и низким уровнем ЭВМ [12]. По мнению сторонников этих подходов, тогда были созданы более качественные средства, обеспечивающие: лучшее выявление ошибок, большую эффективность реализации программ, сокращение набора инструкций, упрощение компиляции, снижение объема требуемой памяти. Ряд компьютеров имеет объектно-ориентированную архитектуру: Burroughs 5000, Plessey 250, Cambridge CAP [13], SWARD [14], Intel 432 [15], Caltech's COM [16], IBM System/38 [17], Rational R1000, BiiN 40 и 60.
С объектно-ориентированной архитектурой тесно связаны объектно-ориентированные операционные системы (ОС). Дейкстра, работая над мультипрограммной системой THE, впервые ввел понятие машины с уровнями состояния в качестве средства построения системы [18]. Среди первых объектно-ориентированных ОС следует отметить: Plessey/System 250 (для мультипроцессора Plessey 250), Hydra (для CMU C.mmp), CALTSS (для CDC 6400), CAP (для компьютера Cambridge CAP), UCLA Secure UNIX (для PDP 11/45 и 11/70), StarOS (для CMU Cm*), Medusa (также для CMU Cm*) и iMAX (для Intel 432) [19]. Следующее поколение операционных систем, таких, как Microsoft Cairo и Taligent Pink, будет, по всей видимости, объектно-ориентированным.
Наиболее значительный вклад в объектный подход внесен объектными и объектно-ориентированными языками программирования. Впервые понятия классов и объектов введены в языке Simula 67. Система Flex и последовавшие за ней диалекты Smalltalk-72, -74, -76 и, наконец, -80, взяв за основу методы Simula, довели их до логического завершения, выполняя все действия на основе классов. В 1970-х годах создан ряд языков, реализующих идею абстракции данных: Alphard, CLU, Euclid, Gypsy, Mesa и Modula. Затем методы, используемые в языках Simula и Smalltalk, были использованы в традиционных языках высокого уровня. Внесение объектно-ориентированного подхода в С привело к возникновению языков C++ и Objective C. На основе языка Pascal возникли Object Pascal, Eiffel и Ada. Появились диалекты LISP, такие, как Flavors, LOOPS и CLOS (Common LISP Object System), с возможностями языков Simula и Smalltalk. Более подробно особенности этих языков изложены в приложении.
Первым, кто указал на необходимость построения систем в виде структурированных абстракций, был Дейкстра. Позднее Парнас ввел идею скрытия информации [20], а в 70-х годах ряд исследователей, главным образом Лисков и Жиль [21], Гуттаг [22], и Шоу [23], разработал механизмы абстрактных типов данных. Хоар дополнил эти подходы теорией типов и подклассов [24].
Развивавшиеся достаточно независимо технологии построения баз данных также оказали влияние на объектный подход [25], в первую очередь благодаря так называемой модели «сущность-отношение» (ER, entity-relationship) [26]. В моделях ER, впервые предложенных Ченом [27], моделирование происходит в терминах сущностей, их атрибутов и взаимоотношений.
Разработчики способов представления данных в области искусственного интеллекта также внесли свой вклад в понимание объектно-ориентированных абстракций. В 1975 г. Мински выдвинул теорию фреймов для представления реальных объектов в системах распознавания образов и естественных языков [28]. Фреймы стали использоваться в качестве архитектурной основы в различных интеллектуальных системах.
Объектный подход известен еще издавна. Грекам принадлежит идея о том, что мир можно рассматривать в терминах как объектов, так и событий. А в XVII веке Декарт отмечал, что люди обычно имеют объектно-ориентированный взгляд на мир [29]. В XX веке эту тему развивала Рэнд в своей философии объективистской эпистемоло-гии [30]. Позднее Мински предложил модель человеческого мышления, в которой разум человека рассматривается как общность различно мыслящих агентов [31]. Он доказывает, что только совместное действие таких агентов приводит к осмысленному поведению человека.

Объектно-ориентированное программирование. Что же такое объектно-ориентированное программирование (object-oriented programming, OOP)? Мы определяем его следующим образом:

Объектно-ориентированное программирование — это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

В данном определении можно выделить три части: 1) OOP использует в качестве базовых элементов *объекты,* а не алгоритмы (иерархия «быть частью», которая была определена в главе 1); 2) каждый объект является *экземпляром* какого-либо определенного *класса;* 3) классы организованы *иерархически* (см. понятие об иерархии «is a» там же). Программа будет объектно-ориентированной только при соблюдении всех трех указанных требований. В частности, программирование, не основанное на иерархических отношениях, не относится к OOP, а называется *программированием на основе абстрактных типов данных.*

В соответствии с этим определением не все языки программирования являются объектно-ориентированными. Страуструп определил так: «если термин *объектно-ориентированный язык* вообще что-либо означает, то он должен означать язык, имеющий средства хорошей поддержки объектно-ориентированного стиля программирования... Обеспечение такого стиля в свою очередь означает, что в языке удобно пользоваться этим стилем. Если написание программ в стиле OOP требует специальных усилий или оно невозможно совсем, то этот язык не отвечает требованиям OOP» [33]. Теоретически возможна имитация объектно-ориентированного программирования на обычных языках, таких, как Pascal и даже COBOL или ассемблер, но это крайне затруднительно. Карделли и Вегнер говорят, что: «язык программирования является объектно-ориентированным тогда и только тогда, когда выполняются следующие условия:

- Поддерживаются объекты, то есть абстракции данных, имеющие интерфейс в виде именованных операций и собственные данные, с ограничением доступа к ним.
- Объекты относятся к соответствующим типам (классам).
- Типы (классы) могут наследовать атрибуты супертипов (суперклассов)» [34].

Поддержка наследования в таких языках означает возможность установления отношения «is-a» («есть», «это есть», « — это»), например, красная роза — это цветок, а цветок — это растение. Языки, не имеющие таких механизмов, нельзя отнести к объектно-ориентированным. Карделли и Вегнер назвали такие языки *объектными,* но не *объектно-ориентированными.* Согласно этому определению объектно-ориентированными языками являются Smalltalk, Object Pascal, C++ и CLOS, а Ada — объектный язык. Но, поскольку объекты и классы являются элементами обеих групп языков, желательно использовать и в тех, и в других методы объектно-ориентированного проектирования.

Объектно-ориентированное проектирование. Программирование прежде всего подразумевает правильное и эффективное использование механизмов конкретных языков программирования. Проектирование, напротив, основное внимание уделяет правильному и эффективному структурированию сложных систем. Мы определяем объектно-ориентированное проектирование следующим образом:

Объектно-ориентированное проектирование — это методология проектирования, соединяющая в себе процесс объектной декомпозиции и приемы представления логической и физической, а также статической и динамической моделей проектируемой системы.

В данном определении содержатся две важные части: объектно-ориентированное проектирование 1) основывается на объектно-ориентированной декомпозиции; 2) использует многообразие приемов представления моделей, отражающих логическую (классы и объекты) и физическую (модули и процессы) структуру системы, а также ее статические и динамические аспекты.

Именно объектно-ориентированная декомпозиция отличает объектно-ориентированное проектирование от структурного; в первом случае логическая структура системы отражается абстракциями в виде классов и объектов, во втором — алгоритмами. Иногда мы будем использовать аббревиатуру *OOD,* object-oriented design, для обозначения метода объектно-ориентированного проектирования, изложенного в этой книге.

Объектно-ориентированный анализ. На объектную модель повлияла более ранняя модель жизненного цикла программного обеспечения. Традиционная техника структурного анализа, описанная в работах Де Марко [35], Иордана [36], Гейна и Сарсона [37], а с уточнениями

для режимов реального времени у Варда и Меллора [38] и Хотли и Пирбхая [39], основана на потоках данных в системе. Объектно-ориентированный анализ (или *00А,* object-oriented analysis) направлен на создание моделей реальной действительности на основе объектно-ориентированного мировоззрения.

Объектно-ориентированный анализ — это методология, при которой требования к системе воспринимаются с точки зрения классов и объектов, выявленных в предметной области.

Как соотносятся OOA, OOD и OOP? На результатах OOA формируются модели, на которых основывается OOD; OOD в свою очередь создает фундамент для окончательной реализации системы с использованием методологии OOP.

2.2. Составные части объектного подхода

Парадигмы программирования

Дженкинс и Глазго считают, что «в большинстве своем программисты используют в работе один язык программирования и следуют одному стилю. Они программируют в парадигме, навязанной используемым ими языком. Часто они оставляют в стороне альтернативные подходы к цели, а следовательно, им трудно увидеть преимущества стиля, более соответствующего решаемой задаче» [40]. Бобров и Сте-тик так определили понятие стиля программирования: «Это способ построения программ, основанный на определенных принципах программирования, и выбор подходящего языка, который делает понятными программы, написанные в этом стиле» [41]. Эти же авторы выявили пять основных разновидностей стилей программирования, которые перечислены ниже вместе с присущими им видами абстракций:

- процедурно-ориентированный   алгоритмы
- объектно-ориентированный   классы и объекты
- логико-ориентированный   цели, часто выраженные в терминах исчисления предикатов
- ориентированный на правила   правила «если-то»
- ориентированный на ограничения   инвариантные соотношения

Невозможно признать какой-либо стиль программирования наилучшим во всех областях практического применения. Например, для проектирования баз знаний более пригоден стиль, ориентированный на правила, а для вычислительных задач — процедурно-ориентированный. По нашему опыту объектно-ориентированный стиль является наиболее приемлемым для широчайшего круга приложений; действительно, эта парадигма часто служит архитектурным фундаментом, на котором мы основываем другие парадигмы.

Каждый стиль программирования имеет свою концептуальную базу. Каждый стиль требует своего умонастроения и способа восприятия решаемой задачи. Для объектно-ориентированного стиля концептуальная база — это *объектная модель.* Она имеет четыре главных элемента:

- абстрагирование
- инкапсуляция
- модульность
- иерархия.

Эти элементы являются *главными* в том смысле, что без любого из них модель не будет объектно-ориентированной. Кроме главных, имеются еще три дополнительных элемента:

- типизация
- параллелизм
- сохраняемость.

Называя их *дополнительными,* мы имеем в виду, что они полезны в объектной модели, но не обязательны.

Без такой концептуальной основы вы можете программировать на языке типа Smalltalk, Object Pascal, C++, CLOS, Eiffel или Ada, но из-под внешней красоты будет выглядывать стиль FORTRAN, Pascal или C. Выразительная способность объектно-ориентированного языка будет либо потеряна, либо искажена. Но еще более существенно, что при этом будет мало шансов справиться со сложностью решаемых задач.

Абстрагирование

Смысл абстрагирования. Абстрагирование является одним из основных методов, используемых для решения сложных задач. Хоар считает, что «абстрагирование проявляется в нахождении сходств между определенными объектами, ситуациями или процессами реального мира, и в принятии решений на основе этих сходств, отвлекаясь на время от имеющихся различий» [42]. Шоу определила это понятие так: «Упрощенное описание или изложение системы, при котором одни свойства и детали выделяются, а другие опускаются. Хорошей является такая абстракция, которая подчеркивает детали, существенные для рассмотрения и использования, и опускает те, которые на данный момент несущественны» [43]. Берзинс, Грей и На-уман рекомендовали, чтобы «идея квалифицировалась как абстракция только, если она может быть изложена, понята и проанализирована независимо от механизма, который будет в дальнейшем принят для ее реализации» [44]. Суммируя эти разные точки зрения, получим следующее определение абстракции:

Абстракция выделяет существенные характеристики некоторого объекта, отличающие его от всех других видов объектов и, таким образом, четко определяет его концептуальные границы с точки зрения наблюдателя.

Абстрагирование концентрирует внимание на внешних особенностях объекта и позволяет отделить самые существенные особенности поведения от несущественных. Абельсон и Суссман назвали такое разделение смысла и реализации *барьером абстракции* [45], который основывается на принципе минимизации связей, когда интерфейс объекта содержит только существенные аспекты поведения и ничего больше [46]. Мы считаем полезным еще один дополнительный принцип, называемый *принципом наименьшего удивления,* согласно которому абстракция должна охватывать все поведение объекта, но не больше и не меньше, и не привносить сюрпризов или побочных эффектов, лежащих вне ее сферы применимости.

Выбор правильного набора абстракций для заданной предметной области представляет собой главную задачу объектно-ориентированного проектирования. Ввиду важности этой темы ей целиком посвящена глава 4.

По мнению Сейдвица и Старка «существует целый спектр абстракций, начиная с объектов, которые почти точно соответствуют реалиям предметной области, и кончая объектами, не имеющими право на существование» [47]. Вот эти абстракции, начиная от наиболее полезных к наименее полезным:

- Абстракция сущности — Объект представляет собой полезную модель некой сущности в предметной области
- Абстракция поведения — Объект состоит из обобщенного множества операций
- Абстракция виртуальной машины — Объект группирует операции, которые либо вместе используются более высоким уровнем управления, либо сами используют некоторый набор операций более низкого уровня
- Произвольная абстракция — Объект включает в себя набор операций, не имеющих друг с другом ничего общего

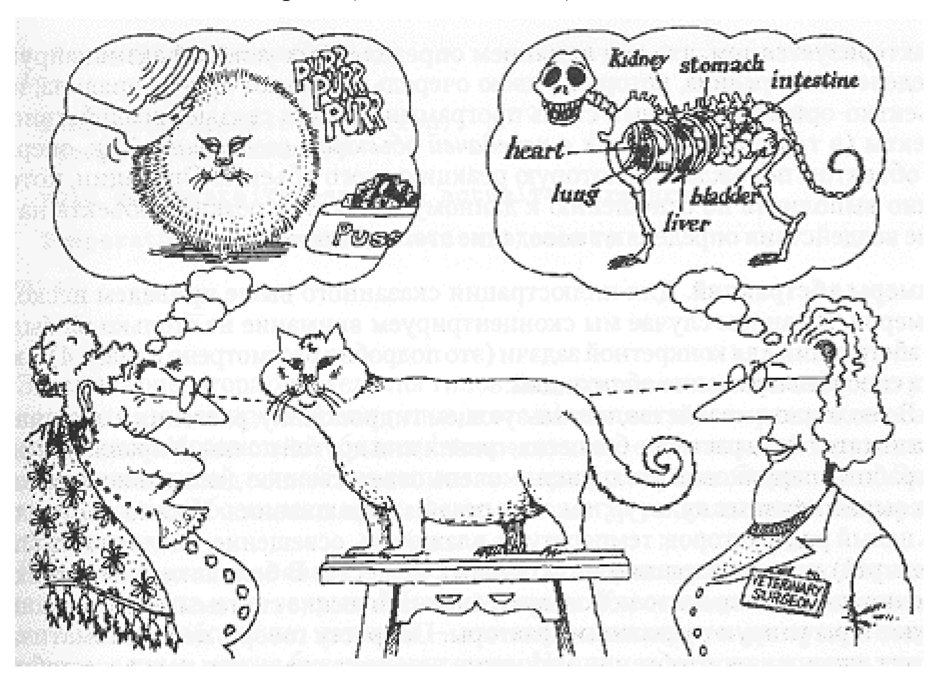
Мы стараемся строить абстракции сущности, так как они прямо соответствуют сущностям предметной области.

Клиентом называется любой объект, использующий ресурсы другого объекта (называемого *сервером).* Мы будем характеризовать поведение объекта услугами, которые он оказывает другим объектам, и операциями, которые он выполняет над другими объектами. Такой подход концентрирует внимание на внешних проявлениях объекта и приводит к идее, которую Мейер назвал *контрактноймоделью* программирования [48]: внешнее проявление объекта рассматривается с точки зрения его контракта с другими объектами, в соответствии с этим должно быть выполнено и его внутреннее устройство (часто во взаимодействии с другими объектами). Контракт фиксирует все обязательства, которые объект-сервер имеет перед объектом-клиентом. Другими словами, этот контракт определяет *ответственность* объекта — то поведение, за которое он отвечает [49].

Каждая операция, предусмотренная этим контрактом, однозначно определяется ее формальными параметрами и типом возвращаемого значения. Полный набор операций, которые

клиент может осуществлять над другим объектом, вместе с правильным порядком, в котором эти операции вызываются, называется *протоколом.* Протокол отражает все возможные способы, которыми объект может действовать или подвергаться воздействию, полностью определяя тем самым внешнее поведение абстракции со статической и динамической точек зрения.

Центральной идеей абстракции является понятие инварианта. *Инвариант* — это некоторое логическое условие, значение которого (истина или ложь) должно



Абстракция фокусируется на существенных с точки зрения наблюдателя характеристиках объекта

сохраняться. Для каждой операции объекта можно задать *предусловия* (инварианты предполагаемые операцией) и *постусловия* (инварианты, которым удовлетворяет операция). Изменение инварианта нарушает контракт, связанный с абстракцией. В частности, если нарушено предусловие, то клиент не соблюдает свои обязательства и сервер не может выполнить свою задачу правильно. Если же нарушено постусловие, то свои обязательства нарушил сервер, и клиент не может более ему доверять. В случае нарушения какого-либо условия возбуждается исключительная ситуация. Как мы увидим далее, некоторые языки имеют средства для работы с исключительными ситуациями: объекты могут возбуждать исключения, чтобы запретить дальнейшую обработку и предупредить о проблеме другие объекты, которые в свою очередь могут принять на себя перехват исключения и справиться с проблемой.

Заметим, что понятия *операция, метод* и *функция-член* происходят от различных традиций программирования (Ada, Smalltalk и C++ соответственно). Фактически они обозначают одно и то же и в дальнейшем будут взаимозаменяемы.

Все абстракции обладают как статическими, так и динамическими свойствами. Например, файл как объект требует определенного объема памяти на конкретном устройстве, имеет имя и содержание. Эти атрибуты являются статическими свойствами. Конкретные же значения каждого из перечисленных свойств динамичны и изменяются в процессе использования объекта: файл можно увеличить или уменьшить, изменить его имя и содержимое. В процедурном стиле программирования действия, изменяющие динамические характеристики объектов, составляют суть программы. Любые события связаны с вызовом подпрограмм и с выполнением операторов. Стиль программирования, ориентированный на правила, характеризуется тем, что под влиянием определенных условий активизируются определенные правила, которые в свою очередь вызывают другие правила, и т. д. Объектно-ориентированный стиль программирования связан с воздействием на объекты (в терминах Smalltalk с *передачей объектам сообщений).* Так, операция над объектом порождает некоторую реакцию этого объекта. Операции, которые можно выполнить по отношению к данному объекту, и реакция объекта на внешние воздействия определяют поведение этого объекта.

Примеры абстракций. Для иллюстрации сказанного выше приведем несколько примеров. В данном случае мы сконцентрируем внимание не столько на выделении абстракций для конкретной задачи (это подробно рассмотрено в главе 4), сколько на способе выражения абстракций.

В тепличном хозяйстве, использующем гидропонику, растения выращиваются на питательном растворе без песка, гравия или другой почвы. Управление режимом работы парниковой установки — очень ответственное дело, зависящее как от вида выращиваемых культур, так и от стадии выращивания. Нужно контролировать целый ряд факторов: температуру, влажность, освещение, кислотность (показатель pH) и концентрацию питательных веществ. В больших хозяйствах для решения этой задачи часто используют автоматические системы, которые контролируют и регулируют указанные факторы. Попросту говоря, цель автоматизации состоит здесь в том, чтобы при минимальном вмешательстве человека добиться соблюдения режима выращивания.

Одна из ключевых абстракций в такой задаче — датчик. Известно несколько разновидностей датчиков. Все, что влияет на урожай, должно быть измерено, так что мы должны иметь датчики температуры воды и воздуха, влажности, pH, освещения и концентрации питательных веществ. С внешней точки зрения датчик температуры — это объект, который способен измерять температуру там, где он расположен. Что такое температура? Это числовой параметр, имеющий ограниченный диапазон значений и определенную точность, означающий число градусов по Фаренгейту, Цельсию или Кельвину. Что такое местоположение датчика? Это некоторое идентифицируемое место в теплице, температуру в котором нам необходимо знать; таких мест, вероятно, немного. Для датчика температуры существенно не столько само местоположение, сколько тот факт, что данный датчик расположен именно в данном месте и это отличает его от других датчиков. Теперь можно задать вопрос о том, каковы обязанности датчика температуры? Мы решаем, что датчик должен знать температуру в своем местонахождении и сообщать ее по запросу. Какие же действия может выполнять по отношению к датчику клиент? Мы принимаем решение о том, что клиент может калибровать датчик и получать от него значение текущей температуры.

Для демонстрации проектных решений будет использован язык C++. Читатели, недостаточно знакомые с этим языком, а также желающие уточнить свои знания по другим объектным и объектно-ориентированным языкам, упоминаемым в этой книге, могут найти их краткие описания с примерами в приложении. Итак, вот описания, задающие абстрактный датчик температуры на C++.

```
// Температура по Фаренгейту
typedef float Temperature;

// Число, однозначно определяющее положение датчика typedef unsigned
int Location;

class TemperatureSensor {
public:
      TemperatureSensor (Location);
      -TemperatureSensor() ;
      void calibrate(Temperature actualTemperature);
      Temperature currentTemperature() const;
private:
…
};
```

Здесь два оператора определения типов **Temperature** и **Location** вводят удобные псевдонимы для простейших типов, и это позволяет нам выражать свои абстракции на языке предметной области.[7] **Temperature** — это числовой тип данных в формате с плавающей точкой

[7] К сожалению, конструкция typedef не определяет нового типа данных и не обеспечивает его защиты. Например, следующее описание в C++:

```
typedef int Count;
```

для записи температур в шкале Фаренгейта. Значения типа Location обозначают места фермы, где могут располагаться температурные датчики.

Класс **TemperatureSensor** — это только спецификация датчика; настоящая его начинка скрыта в его закрытой (private) части. Класс **TemperatureSensor** это еще не объект. Собственно датчики — это его *экземпляры,* и их нужно создать, прежде чем с ними можно будет оперировать. Например, можно написать так:

```
Temperature temperature;

TemperatureSensor greenhouse1Sensor(1);
TemperatureSensor greenhouse2Sensor(2) ;

temperature = greenhouse1Sensor.currentTemperature();
```

Рассмотрим инварианты, связанные с операцией currentTemperature. Предусловие включает предположение, что датчик установлен в правильным месте в теплице, а постусловие — что датчик возвращает значение температуры в градусах Фаренгейта.

До сих пор мы считали датчик пассивным: кто-то должен запросить у него температуру, и тогда он ответит. Однако есть и другой, столь же правомочный подход. Датчик мог бы активно следить за температурой и извещать другие объекты, когда ее отклонение от заданного значения превышает заданный уровень. Абстракция от этого меняется мало: всего лишь несколько иначе формулируется ответственность объекта. Какие новые операции нужны ему в связи с этим? Обычной идиомой для таких случаев является обратный вызов. Клиент предоставляет серверу функцию (функцию обратного вызова), а сервер вызывает ее, когда считает нужным. Здесь нужно написать что-нибудь вроде:

```
class ActiveTemperatureSensor {
public:
      ActiveTemperatureSensor (Location,
            void (*f)(Location, Temperature));
      ~ActiveTemperatureSensor() ;
      void calibrate(Temperature actualTemperature);
      void establishSetpoint(Temperature setpoint,
           Temperature delta);
      Temperature currentTemperature() const;
private:
...
};
```

Новый класс **ActiveTemperatureSensor** стал лишь чуть сложнее, но вполне адекватно выражает новую абстракцию. Создавая экземпляр датчика, мы передаем ему при инициализации не только место, но и указатель на функцию обратного вызова, параметры которой определяют место установки и температуру. Новая функция установки **establishSetpoint** позволяет клиенту изменять порог срабатывания датчика температуры, а ответственность датчика состоит в том, чтобы вызывать функцию обратного вызова каждый раз, когда текущая температура **actualTemperature** отклоняется от **setpoint** больше чем на **delta**. При этом клиенту становится известно место срабатывания и температура в нем, а дальше уже он сам должен знать, что с этим делать.

Заметьте, что клиент по-прежнему может запрашивать температуру по собственной инициативе. Но что если клиент не произведет инициализацию, например, не задаст допустимую температуру? При проектировании мы обязательно должны решить этот вопрос, приняв какое-нибудь разумное допущение: пусть считается, что интервал допустимых изменений температуры бесконечно широк.

Как именно класс **ActiveTemperatureSensor** выполняет свои обязательства, зависит от его внутреннего представления и не должно интересовать внешних клиентов. Это определяется реализацией его закрытой части и функций-членов.

просто вводит синоним для примитивного типа int. Как мы увидим в следующем разделе, другие языки, такие как Ada и Eiffel, имеют более изощренную семантику в отношении строгой типизации базовых типов.

Рассмотрим теперь другой пример абстракции. Для каждой выращиваемой культуры должен быть задан план выращивания, описывающий изменение во времени температуры, освещения, подкормки и ряда других факторов, обеспечивающих высокий урожай. Поскольку такой план является частью предметной области, вполне оправдана его реализация в виде абстракции.

Для каждой выращиваемой культуры существует свой отдельный план, но общая форма планов у всех культур одинакова. Основу плана выращивания составляет таблица, сопоставляющая моментам времени перечень необходимых действий. Например, для некоторой культуры на 15-е сутки роста план предусматривает поддержание в течении 16 часов температуры 78°F, из них 14 часов с освещением, а затем понижение температуры до 65°F на остальное время суток. Кроме того, может потребоваться внесение удобрений в середине дня, чтобы поддержать заданное значение кислотности.

Таким образом, план выращивания отвечает за координацию во времени всех действий, необходимых при выращивании культуры. Наше решение заключается в том, чтобы не поручать абстракции плана само выполнение плана, — это будет обязанностью другой абстракции. Так мы ясно разделим понятия между различными частями системы и ограничим концептуальный размер каждой отдельной абстракции.

С точки зрения интерфейса объекта-плана, клиент должен иметь возможность устанавливать детали плана, изменять план и запрашивать его. Например, объект может быть реализован с интерфейсом «человек-компьютер» и ручным изменением плана. Объект, который содержит детали плана выращивания, должен уметь изменять сам себя. Кроме того, должен существовать объект-исполнитель плана, умеющий читать план. Как видно из дальнейшего описания, ни один объект не обособлен, а все они взаимодействуют для обеспечения общей цели. Исходя из такого подхода, определяются границы каждого объекта-абстракции и протоколы их связи.

На C++ план выращивания будет выглядеть следующим образом. Сначала введем новые типы данных, приближая наши абстракции к словарю предметной области (день, час, освещение, кислотность, концентрация):

```
// Число, обозначающее день года
typedef unsigned int Day;

// Число, обозначающее час дня
typedef unsigned int Hour;

// Булевский тип
enum Lights {OFF, ON};

// Число, обозначающее показатель кислотности в диапазоне от 1 до 14
typedef float  pH;

// Число, обозначающее концентрацию в процентах: от 0 до 100
typedef float Concentration;
```

Далее, в тактических целях, опишем следующую структуру:

```
// Структура, определяющая условия в теплице
struct Condition {
	Temperature temperature;
	Lights lighting;
	pH acidity;
	Concentration concentration;
};
```

Мы использовали структуру, а не класс, поскольку Condition — это просто механическое объединение параметров, без какого-либо внутреннего поведения, и более богатая семантика класса здесь не нужна.

Наконец, вот и план выращивания:

```
class GrowingPlan (
public:
	GrowingPlan (char *name);
	virtual
	~GrowingPlan();
	void clear();
	virtual void establish(Day, Hour, const Condition&);
	const char* name() const;
	const Condition& desiredConditions(Day, Hour) const;
protected:
…
};
```

Заметьте, что мы предусмотрели одну новую обязанность: каждый план имеет имя, и его можно устанавливать и запрашивать. Кроме того заметьте, что операция `establish` описана как virtual для того, чтобы подклассы могли ее переопределять.

В открытую (public) часть описания вынесены конструктор и деструктор объекта (определяющие процедуры его порождения и уничтожения), две процедуры модификации (очистка всего плана clear и определение элементов плана establish) и два селектора-определителя состояния (функции name и `desiredCondition`). Мы опустили в описании закрытую часть класса, заменив ее многоточием, поскольку сейчас нам важны внешние ответственности, а не внутреннее представление класса.

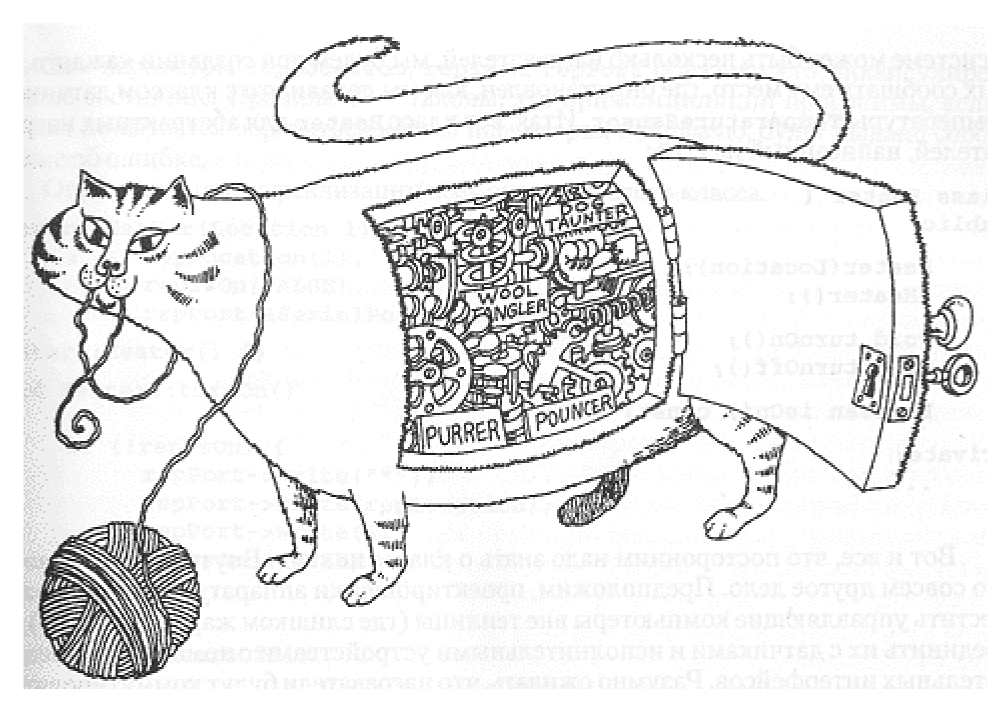
Инкапсуляция

Что это значит? Хотя мы описывали нашу абстракцию `GrowingPlan` как сопоставление действий моментам времени, она не обязательно должна быть реализована буквально как таблица данных. Действительно, клиенту нет никакого дела до реализации класса, который его обслуживает, до тех пор, пока тот соблюдает свои обязательства. На самом деле, абстракция объекта всегда предшествует его реализации. А после того, как решение о реализации принято, оно должно трактоваться как секрет абстракции, скрытый от большинства клиентов. Как мудро замечает Ингалс: «Никакая часть сложной системы не должна зависеть от внутреннего устройства какой-либо другой части» [50]. В то время, как абстракция «помогает людям думать о том, что они делают», инкапсуляция «позволяет легко перестраивать программы» [51].

Абстракция и инкапсуляция дополняют друг друга: абстрагирование направлено на наблюдаемое поведение объекта, а инкапсуляция занимается внутренним устройством. Чаще всего инкапсуляция выполняется посредством скрытия информации, то есть маскировкой всех внутренних деталей, не влияющих на внешнее поведение. Обычно скрываются и внутренняя структура объекта и реализация его методов.

Инкапсуляция, таким образом, определяет четкие границы между различными абстракциями. Возьмем для примера структуру растения: чтобы понять на верхнем уровне действие фотосинтеза, вполне допустимо игнорировать такие подробности, как функции корней растения или химию клеточных стенок. Аналогичным образом при проектировании базы данных принято писать программы так, чтобы они не зависели от физического представления данных; вместо этого сосредотачиваются на схеме, отражающей логическое строение данных [52]. В обоих случаях объекты защищены от деталей реализации объектов более низкого уровня.

Дисков прямо утверждает, что «абстракция будет работать только вместе с инкапсуляцией» [53]. Практически это означает наличие двух частей в классе: интерфейса и реализации. *Интерфейс* отражает внешнее поведение объекта, описывая абстракцию поведения всех объектов данного класса. Внутренняя *реализация* описывает представление этой абстракции и механизмы достижения желаемого поведения объекта. Принцип разделения интерфейса и реализации соответствует сути вещей: в интерфейсной части собрано все, что касается взаимодействия данного объекта с любыми другими объектами; реализация скрывает от других объектов все детали, не имеющие отношения к процессу взаимодействия объектов. Бритон и Парнас назвали такие детали «тайнами абстракции» [54].



Инкапсуляция скрывает детали реализации объекта

Итак, инкапсуляцию можно определить следующим образом:

Инкапсуляция — это процесс отделения друг от друга элементов объекта, определяющих его устройство и поведение; инкапсуляция служит для того, чтобы изолировать контрактные обязательства абстракции от их реализации.

Примеры инкапсуляции. Вернемся к примеру гидропонного тепличного хозяйства. Еще одной из ключевых абстракций данной предметной области является нагреватель, поддерживающий заданную температуру в помещении. Нагреватель является абстракцией низкого уровня, поэтому можно ограничиться всего тремя действиями с этим объектом: включение, выключение и запрос состояния. Нагреватель не должен отвечать за поддержание температуры, это будет поведением более высокого уровня, совместно реализуемым нагревателем, датчиком температуры и еще одним объектом. Мы говорим о поведении *более высокого уровня,* потому что оно основывается на простом поведении нагревателя и датчика, добавляя к ним кое-что еще, а именно *гистерезис* (или запаздывание), благодаря которому можно обойтись без частых включений и выключении нагревателя в состояниях, близких к граничным. Приняв такое решение о разделении ответственности, мы делаем каждую абстракцию более цельной.

Как всегда, начнем с типов.

```
// Булевский тип
enum Boolean {FALSE, TRUE};
```

В дополнение к трем предложенным выше операциям, нужны обычные мета-операции создания и уничтожения объекта (конструктор и деструктор). Поскольку в системе может быть несколько нагревателей, мы будем при создании каждого из них сообщать ему место, где он установлен, как мы делали это с классом датчиков температуры **TemperatureSensor**. Итак, вот класс **Heater** для абстрактных нагревателей, написанный на C++:

```
class Heater {
public:
	Heater(Location);
	~Heater();
	void turnOn();
	void tum0ff();
	Boolean is0n() const;
private:
};
```

Вот и все, что посторонним надо знать о классе **Heater**. Внутренность класса это совсем другое дело. Предположим, проектировщики аппаратуры решили разместить управляющие компьютеры вне теплицы (где слишком жарко и влажно), и соединить их с датчиками и исполнительными устройствами с помощью последовательных интерфейсов. Разумно ожидать, что нагреватели будут коммутироваться с помощью блока реле, а оно будет управляться командами, поступающими через последовательный интерфейс. Скажем, для включения нагревателя передается текстовое имя команды, номер места нагревателя и еще одно число, используемое как сигнал включения нагревателя.

Вот класс, выражающий абстрактный последовательный порт.

```
class SerialPort {
public:
	SerialPort();
	~SerialPort();
	void write(char*);
	void write(int);
	static SerialPort ports[10];
private:
};
```

Экземпляры этого класса будут настоящими последовательными портами, в которые можно выводить строки и числа.

Добавим еще три параметра в класс **Heater**.

```
class Heater {
public:
…
protected:
	const Location repLocation;
	Boolean repIsOn;
	SerialPort* repPort;
};
```

Эти параметры **repLocation**, **repIsOn**, **repPort** образуют его инкапсулированное состояние. Правила С++ таковы, что при компиляции программы, если клиент попытается обратиться к этим параметрам напрямую, будет выдано сообщение об ошибке.

Определим теперь реализации всех операций этого класса.

```
Heater::Heater(Location 1)
	:	repLocation(1),
		repIsOn(FALSE),
		repPort(&SerialPort::ports[l]) {}

Heater::Heater() {}

void Heater::turnOn()
{
	if (!repls0n) {
			repPort->write("*");
			repPort->write(repLocation);
			repPort->write(1);
			repIsOn = TRUE;
		}
}

void Heater::turn0ff()
{
	if (repIsOn) {
			repPort->write("*");
			repPort->write(repLocation);
			repPort->write(0);
```

```
                        repIsOn = FALSE;
                }
}

Boolean Heater::isOn() const
{
        return repIsOn;
}
```

Такой стиль реализации типичен для хорошо структурированных объектно-ориентированных систем: классы записываются экономно, поскольку их специализация осуществляется через подклассы.

Предположим, что по какой-либо причине изменилась архитектура аппаратных средств системы и вместо последовательного порта управление должно осуществляться через фиксированную область памяти. Нет необходимости изменять интерфейсную часть класса — достаточно переписать реализацию. Согласно правилам C++, после этого придется перекомпилировать измененный класс, но не другие объекты, если только они не зависят от временных и пространственных характеристик прежнего кода (что крайне нежелательно и совершенно не нужно).

Обратимся теперь к реализации класса **GrowingPlan**. Как было сказано, это, в сущности, временной график действий. Вероятно, лучшей реализацией его был бы словарь пар время-действие с открытой хеш-таблицей. Нет смысла запоминать действия час за часом, они происходят не так часто, а в промежутках между ними система может интерполировать ход процесса.

Инкапсуляция скроет от посторонних взглядов два секрета: то, что в действительности график использует открытую хеш-таблицу, и то, что промежуточные значения интерполируются. Клиенты вольны думать, что они получают данные из почасового массива значений параметров.

Разумная инкапсуляция локализует те особенности проекта, которые могут подвергнуться изменениям. По мере развития системы разработчики могут решить, что какие-то операции выполняются несколько дольше, чем допустимо, а какие-то объекты занимают больше памяти, чем приемлемо. В таких ситуациях часто изменяют внутреннее представление объекта, чтобы реализовать более эффективные алгоритмы или оптимизировать алгоритм по критерию памяти, заменяя хранение данных вычислением. Важным преимуществом ограничения доступа является возможность внесения изменений в объект без изменения других объектов.

В идеальном случае попытки обращения к данным, закрытым для доступа, должны выявляться во время компиляции программы. Вопрос реализации этих условий для конкретных языков программирования является предметом постоянных обсуждений. Так, Smalltalk обеспечивает защиту от прямого доступа к экземплярам другого класса, обнаруживая такие попытки во время компиляции. В тоже время Object Pascal не инкапсулирует представление класса, так что ничто в этом языке не предохраняет клиента от прямых ссылок на внутренние поля другого объекта. Язык CLOS занимает в этом вопросе промежуточную позицию, возлагая все обязанности по ограничению доступа на программиста. В этом языке все *слоты* могут сопровождаться атрибутами **:reader**, **:writer** и **: accessor**, разрешающими соответственно чтение, запись или полный доступ к данным (то есть и чтение, и запись). При отсутствии атрибутов слот полностью инкапсулирован. По соглашению, признание того, что некоторая величина хранится в слоте, рассматривается как нарушение абстракции, так что хороший стиль программирования на CLOS требует, чтобы при публикации интерфейса класса, документировались бы только имена его функций, а тот факт, что слот имеет функции полного доступа, должен скрываться [55]. В языке C++ управление доступом и видимостью более гибко. Члены класса могут быть отнесены к открытой, закрытой или защищенной частям. Открытая часть доступна для всех объектов; закрытая часть полностью закрыта для других объектов; защищенная часть видна только экземплярам данного класса и его подклассов. Кроме того, в C++ существует понятие «друзей» (friends), для которых открыта закрытая часть.

Скрытие информации — понятие относительное: то, что спрятано на одном уровне абстракции, обнаруживается на другом уровне. Забраться внутрь объектов можно; правда, обычно требуется, чтобы разработчик класса-сервера об этом специально позаботился, а разработчики классов-клиентов не поленились в этом разобраться. Инкапсуляция не спасает от глупости; она,

как отметил Страуструп, «защищает от ошибок, но не от жульничества» [56]. Разумеется, язык программирования тут вообще ни при чем; разве что операционная система может ограничить доступ к файлам, в которых описаны реализации классов. На практике же иногда просто необходимо ознакомиться с реализацией класса, чтобы понять его назначение, особенно, если нет внешней документации.

Модульность

Понятие модульности. По мнению Майерса «Разделение программы на модули до некоторой степени позволяет уменьшить ее сложность... Однако гораздо важнее тот факт, что внутри модульной программы создаются множества хорошо определенных и документированных интерфейсов. Эти интерфейсы неоценимы для исчерпывающего понимания программы в целом» [57]. В некоторых языках программирования, например в Smalltalk, модулей нет, и классы составляют единственную физическую основу декомпозиции. В других языках, включая Object Pascal, C++, Ada, CLOS, модуль — это самостоятельная языковая конструкция. В этих языках классы и объекты составляют логическую структуру системы, они помещаются в *модули,* образующие физическую структуру системы. Это свойство становится особенно полезным, когда система состоит из многих сотен классов.

Согласно Барбаре Лисков «модульность — это разделение программы на фрагменты, которые компилируются по отдельности, но могут устанавливать связи с другими модулями». Мы будем пользоваться определением Парнаса: «Связи между модулями — это их представления друг о друге» [58]. В большинстве языков, поддерживающих принцип модульности как самостоятельную концепцию, интерфейс модуля отделен от его реализации. Таким образом, модульность и инкапсуляция ходят рука об руку. В разных языках программирования модульность поддерживается по-разному. Например, в C++ модулями являются раздельно компилируемые файлы. Для C/C++ традиционным является помещение интерфейсной части модулей в отдельные файлы с расширением .h (так называемые *файлы-заголовки).* Реализация, то есть текст модуля, хранится в файлах с расширением .c (в программах на C++ часто используются расширения .ee, .cp и .cpp). Связь между файлами объявляется директивой макропроцессора #include. Такой подход строится исключительно на соглашении и не является строгим требованием самого языка. В языке Object Pascal принцип модульности формализован несколько строже. В этом языке определен особый синтаксис для интерфейсной части и реализации модуля (unit). Язык Ada идет еще на шаг дальше: модуль (называемый package) также имеет две части — спецификацию и тело. Но, в отличие от Object Pascal, допускается раздельное определение связей с модулями для спецификации и тела пакета. Таким образом, допускается, чтобы тело модуля имело связи с модулями, невидимыми для его спецификации.

Правильное разделение программы на модули является почти такой же сложной задачей, как выбор правильного набора абстракций. Абсолютно прав Зелько-виц, утверждая: «поскольку в начале работы над проектом решения могут быть неясными, декомпозиция на модули может вызвать затруднения. Для хорошо известных приложений (например, создание компиляторов) этот процесс можно стандартизовать, но для новых задач (военные системы или управление космическими аппаратами) задача может быть очень трудной» [59].

Модули выполняют роль физических контейнеров, в которые помещаются определения классов и объектов при логическом проектировании системы. Такая же ситуация возникает у проектировщиков бортовых компьютеров. Логика электронного оборудования может быть построена на основе элементарных схем типа НЕ, И-НЕ, ИЛИ-НЕ, но можно объединить такие схемы в стандартные интегральные схемы (модули), например, серий 7400, 7402 или 7404.

Для небольших задач допустимо описание всех классов и объектов в одном модуле. Однако для большинства программ (кроме самых тривиальных) лучшим решением будет сгруппировать в отдельный модуль логически связанные классы и объекты, оставив открытыми те элементы, которые совершенно необходимо видеть другим модулям. Такой способ разбиения на модули хорош, но его можно довести до абсурда. Рассмотрим, например, задачу, которая выполняется на многопроцессорном оборудовании и требует для координации своей работы механизм передачи сообщений. В больших системах, подобных описываемым в главе 12,



Модульность позволяет хранить абстракции раздельно

вполне обычным является наличие нескольких сотен и даже тысяч видов сообщений. Было бы наивным определять каждый класс сообщения в отдельном модуле. При этом не только возникает кошмар с документированием, но даже просто поиск нужных фрагментов описания становится чрезвычайно труден для пользователя. При внесении в проект изменений потребуется модифицировать и перекомпилировать сотни модулей. Этот пример показывает, что скрытие информации имеет и обратную сторону [60]. Деление программы на модули бессистемным образом иногда гораздо хуже, чем отсутствие модульности вообще.

В традиционном структурном проектировании модульность — это искусство раскладывать подпрограммы по кучкам так, чтобы в одну кучку попадали подпрограммы, использующие друг друга или изменяемые вместе. В объектно-ориентированном программировании ситуация несколько иная: необходимо физически разделить классы и объекты, составляющие логическую структуру проекта.

На основе имеющегося опыта можно перечислить приемы и правила, которые позволяют составлять модули из классов и объектов наиболее эффективным образом. Бритон и Парнас считают, что «конечной целью декомпозиции программы на модули является снижение затрат на программирование за счет независимой разработки и тестирования. Структура модуля должна быть достаточно простой для восприятия; реализация каждого модуля не должна зависеть от реализации других модулей; должны быть приняты меры для облегчения процесса внесения изменений там, где они наиболее вероятны» [61]. Прагматические соображения ставят предел этим руководящим указаниям. На практике перекомпиляция тела модуля не является трудоемкой операцией: заново компилируется только данный модуль, и программа перекомпонуется. Перекомпиляция *интерфейсной* части модуля, напротив, более трудоемка. В строго типизированных языках приходится перекомпилировать интерфейс и тело самого измененного модуля, затем все модули, связанные с данным, модули, связанные с ними, и так далее по цепочке. В итоге для очень больших программ могут потребоваться многие часы на перекомпиляцию (если только среда разработки не поддерживает фрагментарную компиляцию), что явно нежелательно. Поэтому следует стремиться к тому, чтобы интерфейсная часть модулей была возможно более узкой (в пределах обеспечения необходимых связей). Наш стиль программирования требует скрыть все, что только возможно, в реализации модуля. Постепенный перенос описаний из реализации в интерфейсную часть гораздо менее опасен, чем «вычищение» избыточного интерфейсного кода.

Таким образом, программист должен находить баланс между двумя противоположными тенденциями: стремлением скрыть информацию и необходимостью обеспечения видимости тех или иных абстракций в нескольких модулях. Парнас, Клеменс и Вейс предложили следующее правило: «Особенности системы, подверженные изменениям, следует скрывать в отдельных модулях; в качестве межмодульных можно использовать только те элементы, вероятность

изменения которых мала. Все структуры данных должны быть обособлены в модуле; доступ к ним будет возможен для всех процедур этого модуля и закрыт для всех других. Доступ к данным из модуля должен осуществляться только через процедуры данного модуля» [62]. Другими словами, следует стремиться построить модули так, чтобы объединить логически связанные абстракции и минимизировать взаимные связи между модулями. Исходя из этого, приведем определение модульности:

Модульность — это свойство системы, которая была разложена на внутренне связные, но слабо связанные между собой модули.

Таким образом, принципы абстрагирования, инкапсуляции и модульности являются взаимодополняющими. Объект логически определяет границы определенной абстракции, а инкапсуляция и модульность делают их физически незыблемыми.

В процессе разделения системы на модули могут быть полезными два правила. Во-первых, поскольку модули служат в качестве элементарных и неделимых блоков программы, которые могут использоваться в системе повторно, распределение классов и объектов по модулям должно учитывать это. Во-вторых, многие компиляторы создают отдельный сегмент кода для каждого модуля. Поэтому могут появиться ограничения на размер модуля. Динамика вызовов подпрограмм и расположение описаний внутри модулей может сильно повлиять на локальность ссылок и на управление страницами виртуальной памяти. При плохом разбиении процедур по модулям учащаются взаимные вызовы между сегментами, что приводит к потере эффективности кэш-памяти и частой смене страниц.

На выбор разбиения на модули могут влиять и некоторые внешние обстоятельства. При коллективной разработке программ распределение работы осуществляется, как правило, по модульному принципу и правильное разделение проекта минимизирует связи между участниками. При этом более опытные программисты обычно отвечают за интерфейс модулей, а менее опытные — за реализацию. На более крупном уровне такие же соотношения справедливы для отношений между субподрядчиками. Абстракции можно распределить так, чтобы быстро установить интерфейсы модулей по соглашению между компаниями, участвующими в работе. Изменения в интерфейсе вызывают много крика и зубовного скрежета, не говоря уже об огромном расходе бумаги, — все эти факторы делают интерфейс крайне консервативным. Что касается документирования проекта, то оно строится, как правило, также по модульному принципу — модуль служит единицей описания и администрирования. Десять модулей вместо одного потребуют в десять раз больше описаний, и поэтому, к сожалению, иногда требования по документированию влияют на декомпозицию проекта (в большинстве случаев негативно). Могут сказываться и требования секретности: часть кода может быть несекретной, а дру гая — секретной; последняя тогда выполняется в виде отдельного модуля (модулей).

Свести воедино столь разноречивые требования довольно трудно, но главное уяснить: вычленение классов и объектов в проекте и организация модульной структуры — *независимые* действия. Процесс вычленения классов и объектов составляет часть процесса логического проектирования системы, а деление на модули — этап физического проектирования. Разумеется, иногда невозможно завершить логическое проектирование системы, не завершив физическое проектирование, и наоборот. Два этих процесса выполняются итеративно.

Примеры модульности. Посмотрим, как реализуется модульность в гидропонной огородной системе. Допустим, вместо закупки специализированного аппаратного обеспечения, решено использовать стандартную рабочую станцию с графическим интерфейсом пользователя GUI (Graphical User Interface). С помощью рабочей станции оператор может формировать новые планы выращивания, модифицировать имеющиеся планы и наблюдать за их исполнением. Так как абстракция плана выращивания — одна из ключевых, создадим модуль, содержащий все, относящееся к плану выращивания. На C++ нам понадобится примерно такой файл-заголовок (пусть он называется gplan.h).

```
// gplan.h

#ifndef _GPLAN_H
#define _GPLAN_H 1

#include "gtypes.h"
#include "except.h"
```

```
#include "actions.h"

class GrowingPlan ...
class FruitGrowingPlan ...
class GrainGrowingPlan ...

#endif
```

Здесь мы импортируем в файл три других заголовочных файла с определением интерфейсов, на которые будем ссылаться: gtypes.h, except .h и act ions. h. Собственно код классов мы поместим в модуль реализации, в файл с именем gplan.cpp.

Мы могли бы также собрать в один модуль все программы, относящиеся к окнам диалога, специфичным для данного приложения. Этот модуль наверняка будет зависеть от классов, объявленных в gplan.h, и от других файлов-заголовков с описанием классов GUI.

Вероятно, будет много других модулей, импортирующих интерфейсы более низкого уровня. Наконец мы доберемся до главной функции — точки запуска нашей программы операционной системой. При объектно-ориентированном проектировании это скорее всего будет самая малозначительная и неинтересная часть системы, в то время, как в традиционном структурном подходе головная функция — это краеугольный камень, который держит все сооружение. Мы полагаем, что объектно-ориентированный подход более естественен, поскольку, как замечает Мейер, «на практике программные системы предлагают некоторый набор услуг. Сводить их к одной функции можно, но противоестественно... Настоящие системы не имеют верхнего уровня» [63].

Иерархия

Что такое иерархия? Абстракция — вещь полезная, но всегда, кроме самых простых ситуаций, число абстракций в системе намного превышает наши умственные возможности. Инкапсуляция позволяет в какой-то степени устранить это препятствие, убрав из поля зрения внутреннее содержание абстракций. Модульность также упрощает задачу, объединяя логически связанные абстракции в группы. Но этого оказывается недостаточно.

Значительное упрощение в понимании сложных задач достигается за счет образования из абстракций иерархической структуры. Определим иерархию следующим образом:

Иерархия — это упорядочение абстракций, расположение их по уровням.

Основными видами иерархических структур применительно к сложным системам являются структура классов (иерархия «is-a») и структура объектов (иерархия «part of»).

Примеры иерархии: одиночное наследование. Важным элементом объектно-ориентированных систем и основным видом иерархии «is-a» является упоминавшаяся выше концепция наследования. Наследование означает такое отношение между классами (отношение родитель/потомок), когда один класс заимствует структурную или функциональную часть одного или нескольких других классов (соответственно, *одиночное* и *множественное наследование).* Иными словами, наследование создает такую иерархию абстракций, в которой подклассы наследуют строение от одного или нескольких суперклассов. Часто подкласс достраивает или переписывает компоненты вышестоящего класса.

Семантически, наследование описывает отношение типа «is-a». Например, медведь есть млекопитающее, дом есть недвижимость и «быстрая сортировка» есть сортирующий алгоритм. Таким образом, наследование порождает иерархию «обобщение-специализация», в которой подкласс представляет собой специализированный частный случай своего суперкласса. «Лакмусовая бумажка» наследования — обратная проверка; так, если в не есть A, то в не стоит производить от A.

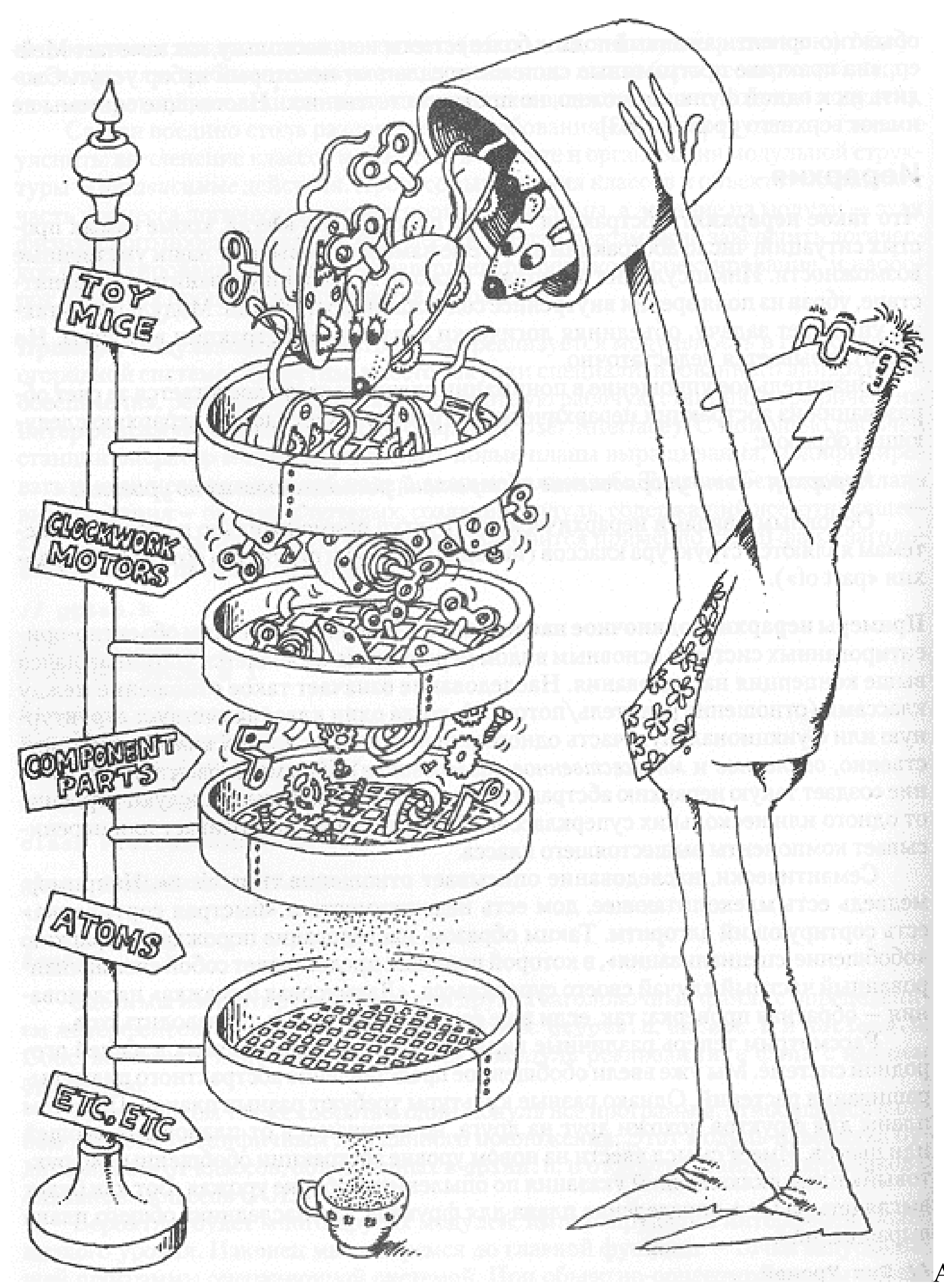
Рассмотрим теперь различные виды растений, выращиваемых в нашей огородной системе. Мы уже ввели обобщенное представление абстрактного плана выращивания растений. Однако разные культуры требуют разных планов. При этом планы для фруктов похожи друг на друга, но отличаются от планов для овощей или цветов. Имеет смысл ввести на новом уровне абстракции обобщенный «фруктовый» план, включающий указания по опылению и сборке урожая. Вот как будет выглядеть на C++ определение плана для фруктов, как наследника общего плана выращивания.

```
// Тип Урожай
typedef unsigned int Yield;
class FruitGrowingPlan : public GrowingPlan {
public:
      FruitGrowingPlan(char* name);
      virtual ~FruitGrowingPlan();

      virtual void establish(Day, Hour, Condition&);
      void scheduleHarvest(Day, Hour);

      Boolean isHarvested() const;
      unsigned daysUntilHarvest() const;
      Yield estimatedYield() const;

protected:
      Boolean repHarvested;
      Yield repYield;
};
```



Абстракции образуют иерархию

Это означает, что план выращивания фруктов **FruitGrowingPlan** является разновидностью плана выращивания **Growingplan**. В него добавлены параметры **repHarvested** и **repYield**, определены четыре новые функции и переопределена функция establish. Теперь мы могли бы продолжить специализацию — например, определить на базе «фруктового» плана «яблочный» класс **AppleGrowingPlan**.

В наследственной иерархии общая часть структуры и поведения сосредоточена в наиболее общем суперклассе. По этой причине говорят о наследовании, как об иерархии *обобщение-специализация.* Суперклассы при этом отражают наиболее общие, а подклассы — более специализированные абстракции, в которых члены суперкласса могут быть дополнены, модифицированы и даже скрыты. Принцип наследования позволяет упростить выражение абстракций, делает проект менее громоздким и более выразительным. Кокс пишет: «В отсутствие наследования каждый класс становится самостоятельным блоком и должен разрабатываться "с нуля". Классы лишаются общности, поскольку каждый программист реализует их по-своему. Стройность системы достигается тогда только за счет дисциплинированности программистов.

Наследование позволяет вводить в обращение новые программы, как мы обучаем новичков новым понятиям — сравнивая новое с чем-то уже известным» [64].

Принципы абстрагирования, инкапсуляции и иерархии находятся между собой в некоем здоровом конфликте. Данфорт и Томлинсон утверждают: «Абстрагирование данных создает непрозрачный барьер, скрывающий состояние и функции объекта; принцип наследования требует открыть доступ и к состоянию, и к функциям объекта для производных объектов» [65]. Для любого класса обычно существуют два вида клиентов: объекты, которые манипулируют с экземплярами данного класса, и подклассы-наследники. Лисков поэтому отмечает, что существуют три способа нарушения инкапсуляции через наследование: «подкласс может получить доступ к переменным экземпляра своего суперкласса, вызвать закрытую функцию и, наконец, обратиться напрямую к суперклассу своего суперкласса» [66]. Различные языки программирования по-разному находят компромисс между наследованием и инкапсуляцией; наиболее гибким в этом отношении является C++. В нем интерфейс класса может быть разделен на три части: закрытую (private), видимую только для самого класса; защищенную (protected), видимую также и для подклассов; и открытую (public), видимую для всех.

Примеры иерархии: множественное наследование. В предыдущем примере рассматривалось одиночное наследование, когда подкласс **`FruitGrowingPlan`** был создан только из одного суперкласса **`Growingplan`**. В ряде случаев полезно реализовать наследование от нескольких суперклассов. Предположим, что нужно определить класс, представляющий разновидности растений.

```
class Plant {
public:
	Plant(char* name, char* species);
	virtual ~Plant();
	void setDatePlanted(Day);
	virtual establishGrowingConditions(const Condition&);
	const char* name() const;
	const char* species() const;
	Day datePlantedt) const;
protected:
	char* repName;
	char* repSpecies;
	Day repPlanted;
private:
…
};
```

Каждый экземпляр класса **`plant`** будет содержать имя, вид и дату посадки. Кроме того, для каждого вида растений можно задавать особые оптимальные условия выращивания. Мы хотим, чтобы эта функция переопределялась подклассами, поэтому она объявлена *виртуальной* при реализации в C++. Три параметра объявлены как защищенные, то есть они будут доступны и классу, и подклассам (закрытая часть спецификации доступна только самому классу).

Изучая предметную область, мы приходим к выводу, что различные группы культивируемых растений — цветы, фрукты и овощи, — имеют свои особые свойства, существенные для технологии их выращивания. Например, для цветов важно знать времена цветения и созревания семян. Аналогично, время сбора урожая важно для абстракций фруктов и овощей. Создадим два новых класса — цветы (**`Flower`**) и фрукты-овощи (**`FruitVegetable`**); они оба наследуют от класса Plant. Однако некоторые цветочные растения имеют плоды! Для этой абстракции придется создать третий класс, FlowerFruitVegetable, который будет наследовать от классов Flower и FruitVegetablePlant.

Чтобы не было избыточности, в данном случае очень пригодится множественное наследование. Сначала давайте опишем отдельно цветы и фрукты-овощи.

```
class FlowerMixin {
public:
	PlowerMixin(Day timeToFlower, Day timeToSeed);
	virtual ~`````FlowerMixin() ;
	Day timeToFlower() const;
```

```
	Day timeToSeed() const;
protected:
…
};

class FruitVegetableMixin {
public:
	FruitVegetableMixin(Day timeToHarvest) ;
	virtual ~FruitVegetableMixin() ;
	Day timeToHarvest() const;
protected:
…
};
```

Мы намеренно описали эти два класса без наследования. Они ни от кого не наследуют и специально предназначены для того, чтобы их *подмешивали* (откуда и имя Mixin) к другим классам. Например, опишем розу:

```
class Rose : public Plant, public FlowerMixin...
```

А вот морковь:

```
class Carrot : public Plant, public FruiteVegetableMixin {};
```

В обоих случаях классы наследуют от двух суперклассов: экземпляры подкласса Rose включают структуру и поведение как из класса Plant, так и из класса FlowerMixin. И вот теперь определим вишню, у которой товаром являются как цветы, так и плоды:

```
class Cherry : public Plant,
               public FlowerMixin,
               FruitVegetableMixin...
```

Множественное наследование — вещь нехитрая, но оно осложняет реализацию языков программирования. Есть две проблемы — конфликты имен между различными суперклассами и повторное наследование. Первый случай, это когда в двух или большем числе суперклассов определено поле или операция с одинаковым именем. В C++ этот вид конфликта должен быть явно разрешен вручную, а в Smalltalk берется то, которое встречается первым. Повторное наследование, это когда класс наследует двум классам, а они порознь наследуют одному и тому же четвертому. Получается ромбическая структура наследования и надо решить, должен ли самый нижний класс получить одну или две отдельные копии самого верхнего класса? В некоторых языках повторное наследование запрещено, в других конфликт решается «волевым порядком», а в C++ это оставляется на усмотрение программиста. Виртуальные базовые классы используются для запрещения дублирования повторяющихся структур, в противном случае в подклассе появятся копии полей и функций и потребуется явное указание происхождения каждой из копий.

Множественным наследованием часто злоупотребляют. Например, сладкая вата — это частный случай сладости, но никак не ваты. Применяйте ту же «лакмусовую бумажку»: если В не есть А, то ему не стоит наследовать от А. Часто плохо сформированные структуры множественного наследования могут быть сведены к единственному суперклассу плюс агрегация других классов подклассом.

Примеры иерархии: агрегация. Если иерархия «is a» определяет отношение «обобщение/специализация», то отношение «part of» (часть) вводит иерархию агрегации. Вот пример.

```
class Garden {
public:
	Garden();
	virtual ~Garden();
protected:
	Plant* repPlants[100];
	GrowingPlan repPlan;
};
```

Это — абстракция огорода, состоящая из массива растений и плана выращивания.

Имея дело с такими иерархиями, мы часто говорим об уровнях абстракции, которые впервые предложил Дейкстра [67]. В иерархии классов вышестоящая абстракция является обобщением, а нижестоящая — специализацией. Поэтому мы говорим, что класс **Flower** находится на более высоком уровне абстракции, чем класс **Plant**. В иерархии «part of» класс

находится на более высоком уровне абстракции, чем любой из использовавшихся при его реализации. Так класс **`Garden`** стоит на более высоком уровне, чем класс **`Plant`**.

Агрегация есть во всех языках, использующих структуры или записи, состоящие из разнотипных данных. Но в объектно-ориентированном программировании она обретает новую мощь: агрегация позволяет физически сгруппировать логически связанные структуры, а наследование с легкостью копирует эти общие группы в различные абстракции.

В связи с агрегацией возникает проблема владения, или принадлежности объектов. В нашем абстрактном огороде одновременно растет много растений, и от удаления или замены одного из них огород не становится другим огородом. Если мы уничтожаем огород, растения остаются (их ведь можно пересадить). Другими словами, огород и растения имеют свои отдельные и независимые сроки жизни; мы достигли этого благодаря тому, что огород содержит не сами объекты **`Plant`**, а указатели на них. Напротив, мы решили, что объект **`GrowingPlan`** внутренне связан с объектом **`Garden`** и не существует независимо. План выращивания физически содержится в каждом экземпляре огорода и погибает вместе с ним. Подробнее про семантику владения мы будем говорить в следующей главе.

Типизация

Что такое типизация? Понятие *типа* взято из теории абстрактных типов данных. Дойч определяет тип, как «точную характеристику свойств, включая структуру и поведение, относящуюся к некоторой совокупности объектов» [68]. Для наших целей достаточно считать, что термины *тип* и *класс* взаимозаменяемы.[8] Тем не менее, типы стоит обсудить отдельно, поскольку они выставляют смысл абстрагирования в совершенно другом свете. В частности, мы утверждаем, что:

Типизация — это способ защититься от использования объектов одного класса вместо другого, или по крайней мере управлять таким использованием.

Типизация заставляет нас выражать наши абстракции так, чтобы язык программирования, используемый в реализации, поддерживал соблюдение принятых проектных решений. Вегнер замечает, что такой способ контроля существенен для программирования «в большом» [70].

Идея согласования типов занимает в понятии типизации центральное место. Например, возьмем физические единицы измерения [71]. Деля расстояние на время, мы ожидаем получить скорость, а не вес. В умножении температуры на силу смысла нет, а в умножении расстояния на силу — есть. Все это примеры сильной типизации, когда прикладная область накладывает правила и ограничения на использование и сочетание абстракций.

Примеры сильной и слабой типизации. Конкретный язык программирования может иметь сильный или слабый механизм типизации, и даже не иметь вообще никакого, оставаясь объектно-ориентированным. Например, в Eiffel соблюдение правил использования типов контролируется непреклонно, — операция не может быть применена к объекту, если она не зарегистрирована в его классе или суперклассе. В сильно типизированных языках нарушение согласования типов может быть обнаружено во время трансляции программы. С другой стороны, в Smalltalk типов нет: во время исполнения любое сообщение можно послать любому объекту, и если класс объекта (или его надкласс) не понимает сообщение, то генерируется сообщение об ошибке. Нарушение согласования типов может не обнаружиться во время трансляции и обычно проявляется как ошибка исполнения. C++ тяготеет к сильной типизации, но в этом языке правила типизации можно игнорировать или подавить полностью.

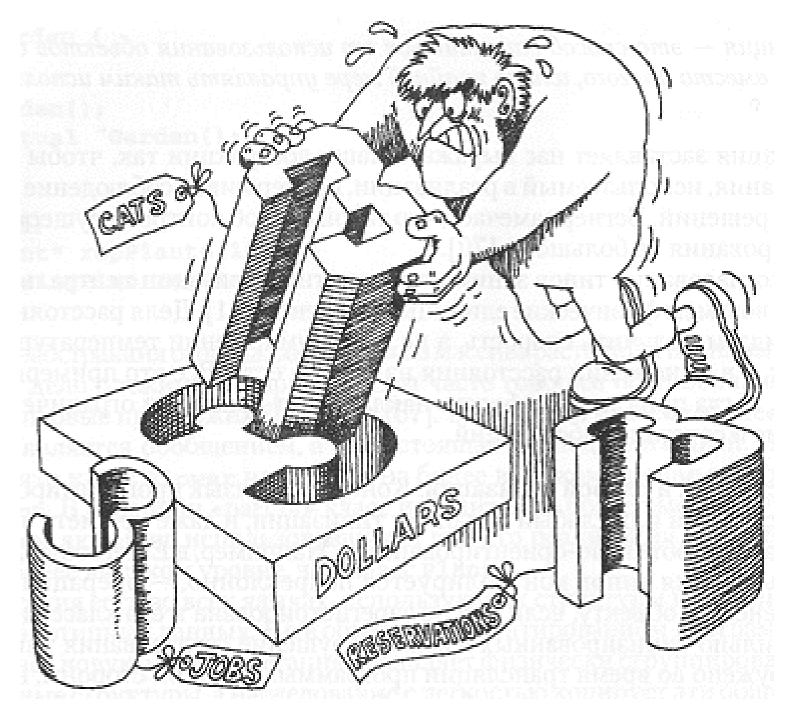
Рассмотрим абстракцию различных типов емкостей, которые могут использоваться в нашей теплице. Вероятно, в ней есть емкости для воды и для минеральных удобрений; хотя первые предназначены для жидкостей, а вторые для сыпучих веществ, они имеют достаточно много общего, чтобы устроить иерархию классов. Начнем с типов.

```
// Число, обозначающее уровень от 0 до 100 процентов
```

[8] Тип и класс не вполне одно и то же; в некоторых языках их различают. Например, ранние версии языка Trellis/Owl разрешали объекту иметь и класс, и тип. Даже в Smalltalk объекты классов SmallInteger, LargeNegativeInteger, LargePositiveInteger ОТНОСЯТСЯ К одному типу Integer, хотя и к разным классам [69]. Большинству смертных различать типы и классы просто противно и бесполезно. Достаточно сказать, что класс реализует понятие типа.

```
typedef float Level;
```

Операторы typedef в C++ не вводят новых типов. В частности, и Level и Concentration — на самом деле другие названия для float, и их можно свободно смешивать в вычислениях. В этом смысле C++ имеет слабую типизацию: значения примитивных типов, таких, как int или float неразличимы в пределах данного типа. Напротив, Ada и Object Pascal предоставляют сильную типизацию для примитивных типов. В Ada можно объявить самостоятельным типом интервал значений или подмножество с ограниченной точностью.



Строгая типизация предотвращает смешивание абстракций

Построим теперь иерархию классов для емкостей:

```
class StorageTank {
public:
	StorageTank();
	virtual ~StorageTank();
	virtual void fill();
	virtual void startDraining() ;
	virtual void stopDraining();
	Boolean isEmpty() const;
	Level level() const;
protected:
…
};

class WaterTank : public StorageTank {
public:
	WaterTank();
	virtual ~WaterTank() ;
	virtual void fill();
	virtual void startDraining();
	virtual void stopDraining();
	void startHeating();
	void stopHeating();
	Temperature currentTemperature() const;
protected:
…
};
```

```
class NutrientTank : public StorageTank (
public:
      NutrientTank() ;
      virtual ~NutrientTank() ;
      virtual void startDrainingt();
      virtual void stopDraining();
protected:
…
};
```

Класс StorageTank — это базовый класс иерархии. Он обеспечивает структуру и поведение общие для всех емкостей: возможность их наполнять или опустошать. Классы WaterTank (емкость для воды) и NutrientTank (для удобрений) наследуют свойства StorageTank, частично переопределяют их и добавляют кое-что свое: например, класс WaterTank вводит новое поведение, связанное с температурой.

Предположим, что мы имеем следующие описания:

```
StorageTank si, s2;
WaterTank w;
NutrientTank n;
```

Заметьте, переменные такие как s1, s2, w или n — это не экземпляры соответствующих классов. На самом деле, это просто имена, которыми мы обозначаем объекты соответствующих классов: когда мы говорим «объект s1» мы на самом деле имеем ввиду экземпляр StorageTank, обозначаемый переменной s1. Мы вернемся к этому тонкому вопросу в следующей главе.

При проверке типов у классов, C++ типизирован гораздо строже. Под этим понимается, что выражения, содержащие вызовы операций, проверяются на согласование типов во время компиляции. Например, следующее правильно:

```
Level 1 = s1.level ();
w.startDrainingt();
n.stopDraining();
```

Действительно, такие селекторы есть в классах, к которым принадлежат соответствующие переменные. Напротив, следующее неправильно и вызовет ошибку компиляции:

```
sl.startHeating(); // Неправильно
n.stopHeating(); // Неправильно
```

Таких функций нет ни в самих классах, ни в их суперклассах. Но следующее

```
n.fill();
```

совершенно правильно: функции fill нет в определении NutrientTank, но она есть в вышестоящем классе.

Итак, сильная типизация заставляет нас соблюдать правила использования абстракций, поэтому она тем полезнее, чем больше проект. Однако у нее есть и теневая сторона. А именно, даже небольшие изменения в интерфейсе класса требуют перекомпиляции всех его подклассов. Кроме того, не имея параметризованных классов, о которых речь пойдет в главах 3 и 9, трудно представить себе, как можно было бы создать собрание разнородных объектов. Предположим, что мы хотим ввести абстракцию инвентарного списка, в котором собирается все имущество, связанное с теплицей. Обычная для С идиома применима и в C++: нужно использовать класс-контейнер, содержащий указатели на void, то есть на объекты произвольного типа.

```
class Inventory {
public:
      Inventory();
      ~Inventory();
      void add(void*) ;
      void remove(void*);
      void* mostRecent() const;
      void apply(Boolean (*)(void*));
private:
…
};
```

Операция apply — это так называемый итератор, который позволяет применить какую-либо операцию ко всем объектам в списке. Подробнее об итераторах см. в следующей главе.

Имея экземпляр класса Inventory, мы можем добавлять и уничтожать указатели на объекты любых классов. Но эти действия не безопасны с точки зрения типов — в списке могут оказаться как осязаемые объекты (емкости), так и неосязаемые (температура или план выращивания), что нарушает нашу абстракцию материального учета. Более того, мы могли бы внести в список объекты классов WaterTank и TemperatureSensor, и по неосторожности ожидая от функции mostRecent объекта класса WaterTank получить StorageTank.

Вообще говоря, у этой проблемы есть два общих решения. Во-первых, можно сделать контейнерный класс, безопасный с точки зрения типов. Чтобы не манипулировать с нетипизированными указателями void, мы могли бы определить инвентаризационный класс, который манипулирует только с объектами класса TangibleAsset (осязаемого имущества), а этот класс будет подмешиваться ко всем классам, такое имущество представляющим, например, к WaterTank, но не к GrowingPlan. Тем самым можно отсечь проблему первого рода, когда неправомочно смешиваются объекты разных типов. Во-вторых, можно ввести проверку типов в ходе выполнения, для того, чтобы знать, с объектом какого типа мы имеем дело в данный момент. Например, в Smalltalk можно запрашивать у объектов их класс. В C++ такая возможность не входила в стандарт до недавнего времени, хотя на практике, конечно, можно ввести в базовый класс операцию, возвращающую код класса (строку или значение перечислимого типа). Однако для этого надо иметь очень серьезные причины, поскольку проверка типа в ходе выполнения ослабляет инкапсуляцию. Как будет показано в следующем разделе, необходимость проверки типа можно смягчить, используя полиморфные операции.

В языках с сильной типизацией гарантируется, что все выражения будут согласованы по типу. Что это значит, лучше пояснить на примере. Следующие присваивания допустимы:

```
s1 = s2;
s1 = w;
```

Первое присваивание допустимо, поскольку переменные имеют один и тот же класс, а второе — поскольку присваивание идет снизу вверх по типам. Однако во втором случае происходит потеря информации (известная в C++ как «проблема срезки»), так как класс переменной **w**, **WaterTank**, семантически богаче, чем класс переменной **s1**, то есть **StorageTank**.

Следующие присваивания неправильны:

```
w = s1; // Неправильно
w = n; // Неправильно
```

В первом случае неправильность в том, что присваивание идет сверху вниз по иерархии, а во втором классы даже не находятся в состоянии подчиненности.

Иногда необходимо преобразовать типы. Например, посмотрите на следующую функцию:

```
void checkLevel(const StorageTank& s);
```

Мы можем привести значение вышестоящего класса к подклассу в том и только в том случае, если фактическим параметром при вызове оказался объект класса WaterTank. Или вот еще случай:

```
if (((WaterTank&)s).currentTemperature() < 32.0) ...
```

Это выражение согласовано по типам, но не безопасно. Если при выполнении программы вдруг окажется, что переменная **s** обозначала объект класса **NutrientTank**, приведение типа даст непредсказуемый результат во время исполнения. Вообще говоря, преобразований типа надо избегать, поскольку они часто представляют собой нарушение принятой системы абстракций.

Теслер отметил следующие важные преимущества строго типизированных языков:

• «Отсутствие контроля типов может приводить к загадочным сбоям в программах во время их выполнения.

• В большинстве систем процесс редактирование-компиляция-отладка утомителен, и раннее обнаружение ошибок просто незаменимо.

• Объявление типов улучшает документирование программ.

• Многие компиляторы генерируют более эффективный объектный код, если им явно известны типы» [72].

Языки, в которых типизация отсутствует, обладают большей гибкостью, но даже в таких языках, по мнению Борнинга и Ингалса: «Программисты обычно знают, какие объекты ожидаются в качестве аргументов и какие будут возвращаться» [73]. На практике, особенно при

программировании «в большом», надежность языков со строгой типизацией с лихвой компенсирует некоторую потерю в гибкости по сравнению с нетипизированными языками.

Примеры типизации: статическое и динамическое связывание. Сильная и статическая типизация — разные вещи. Строгая типизация следит за соответствием типов, а статическая типизация (иначе называемая *статическим* или *ранним связыванием)* определяет время, когда имена связываются с типами. Статическая связь означает, что типы всех переменных и выражений известны во время компиляции;

динамическое связывание (называемое также *поздним связыванием)* означает, что типы неизвестны до момента выполнения программы. Концепции типизации и связывания являются независимыми, поэтому в языке программирования может быть: типизация — сильная, связывание — статическое (Ada), типизация — сильная, связывание — динамическое (C++, Object Pascal), или и типов нет, и связывание динамическое (Smalltalk). Язык CLOS занимает промежуточное положение между C++ и Smalltalk: определения типов, сделанные программистом, могут быть либо приняты во внимание, либо не приняты.

Прокомментируем это понятие снова примером на C++. Вот «свободная», то есть не входящая в определение какого-либо класса, функция:[9]

```
void balanceLevels(StorageTank& s1, StorageTank& s2);
```

Вызов этой функции с экземплярами класса StorageTank или любых его подклассов в качестве параметров будет согласован по типам, поскольку тип каждого фактического параметра происходит в иерархии наследования от базового класса StorageTank.

При реализации этой функции мы можем иметь что-нибудь вроде:

```
if (s1.level() > s2.level ()) s2.fill();
```

В чем особенность семантики при использовании селектора level? Он определен только в классе StorageTank, поэтому, независимо от классов объектов, обозначаемых переменными в момент выполнения, будет использована одна и та же унаследованная ими функция. Вызов этой функции статически связан при компиляции — мы точно знаем, какая операция будет запущена.

Иное дело fill. Этот селектор определен в StorageTank и переопределен в WaterTank, поэтому его придется связывать динамически. Если при выполнении переменная s2 будет класса WaterTank, то функция будет взята из этого класса, а если — NutrientTank, то из StorageTank. В C++ есть специальный синтаксис для явного указания источника; в нашем примере вызов fill будет разрешен, соответственно, как WaterTank::fill или StorageTank::fill.[10]

Это особенность называется *полиморфизмом:* одно и то же имя может означать объекты разных типов, но, имея общего предка, все они имеют и общее подмножество операций, которые можно над ними выполнять [74]. Противоположность полиморфизму называется *мономорфизмом;* он характерен для языков с сильной типизацией и статическим связыванием (Ada).

Полиморфизм возникает там, где взаимодействуют наследование и динамическое связывание. Это одно из самых привлекательных свойств объектно-ориентированных языков (после поддержки абстракции), отличающее их от традиционных языков с абстрактными типами данных. И, как мы увидим в следующих главах, полиморфизм играет очень важную роль в объектно-ориентированном проектировании.

Параллелизм

Что такое параллелизм? Есть задачи, в которых автоматические системы должны обрабатывать много событий одновременно. В других случаях потребность в вычислительной мощности превышает ресурсы одного процессора. В каждой из таких ситуаций естественно использовать несколько компьютеров для решения задачи или задействовать многозадачность на многопроцессорном компьютере. Процесс *(поток управления)* — это фундаментальная единица действия в системе. Каждая программа имеет по крайней мере один поток управления, параллельная система имеет много таких потоков: век одних недолог, а другие живут в течении всего сеанса работы системы. Реальная параллельность достигается только на многопроцессорных

---

[9] Свободная функция — функция, не входящая ни в какой класс. В чисто объектно-ориентированных языках, типа Smalltalk, свободных процедур не бывает, каждая операция связана с каким-нибудь классом.
[10] Так синтаксис C++ определяет явную квалификацию имени.

системах, а системы с одним процессором имитируют параллельность за счет алгоритмов разделения времени.

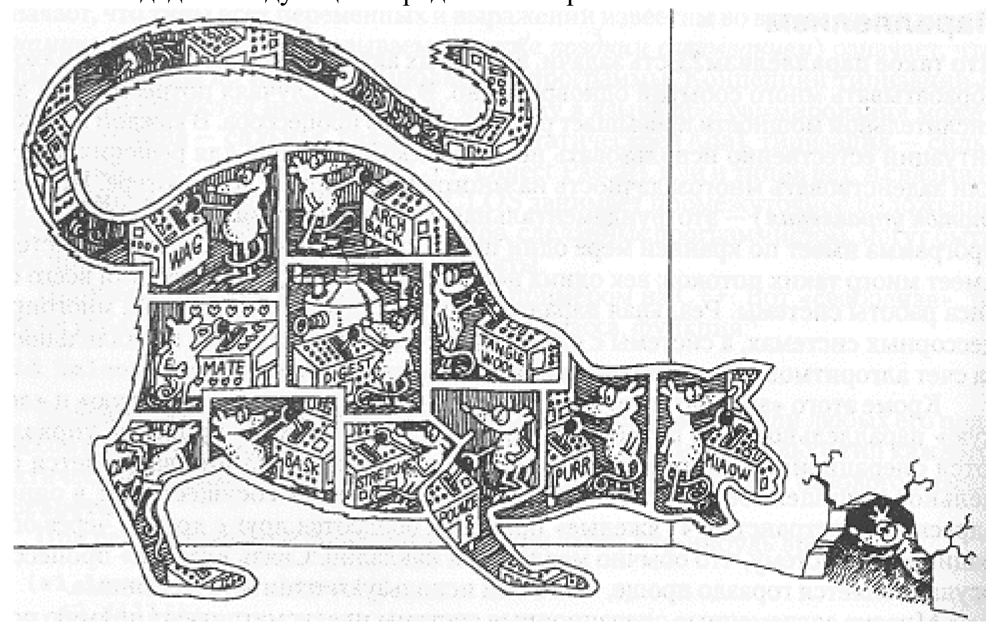
Кроме этого «аппаратного» различия, мы будем различать «тяжелую» и «легкую» параллельность по потребности в ресурсах. «Тяжелые» процессы управляются операционной системой независимо от других, и под них выделяется отдельное защищенное адресное пространство. «Легкие» сосуществуют в одном адресном пространстве. «Тяжелые» процессы общаются друг с другом через операционную систему, что обычно медленно и накладно. Связь «легких» процессов осуществляется гораздо проще, часто они используют одни и те же данные.

Многие современные операционные системы предусматривают прямую поддержку параллелизма, и это обстоятельство очень благоприятно сказывается на возможности обеспечения параллелизма в объектно-ориентированных системах. Например, системы UNIX предусматривают системный вызов *fork,* который порождает новый процесс. Системы Windows NT и OS/2 — многопоточные; кроме того они обеспечивают программные интерфейсы для создания процессов и манипулирования с ними.

Лим и Джонсон отмечают, что «возможности проектирования параллельности в объектно-ориентированных языках не сильно отличаются от любых других, — на нижних уровнях абстракции параллелизм и ООР развиваются совершенно независимо. С ООР или без, все традиционные проблемы параллельного программирования сохраняются» [75]. Действительно, создавать большие программы и так непросто, а если они еще и параллельные, то надо думать о возможном простое одного из потоков, неполучении данных, взаимной блокировке и т. д.

К счастью, как отмечают те же авторы далее: «на верхних уровнях ООР упрощает параллельное программирование для рядовых разработчиков, пряча его в по-вторноиспользуемые абстракции» [76]. Блэк и др. сделали следующий вывод: «объектная модель хороша для распределенных систем, поскольку она неявно разбивает программу на (1) распределенные единицы и (2) сообщающиеся субъекты» [77].

В то время, как объектно-ориентированное программирование основано на абстракции, инкапсуляции и наследовании, параллелизм главное внимание уделяет абстрагированию и синхронизации процессов [78]. Объект есть понятие, на котором эти две точки зрения сходятся: каждый объект (полученный из абстракции реального мира) может представлять собой отдельный поток управления (абстракцию процесса). Такой объект называется *активным.* Для систем, построенных на основе OOD, мир может быть представлен, как совокупность взаимодействующих объектов, часть из которых является активной и выступает в роли независимых вычислительных центров. На этой основе дадим следующее определение параллелизма:



Параллелизм позволяет различным объектам действовать одновременно

Параллелизм — это свойство, отличающее активные объекты от пассивных.

Примеры параллелизма. Ранее мы обзавелись классом ActiveTemperatureSensor, поведение которого предписывает ему периодически измерять температуру и обращаться к

известной ему функции вызова, когда температура отклоняется на некоторую величину от установленного значения. Как он будет это делать, мы в тот момент не объяснили. При всех секретах реализации понятно, что это — активный объект и, следовательно, без параллелизма тут не обойтись. В объектно-ориентированном проектировании есть три подхода к параллелизму.

Во-первых, параллелизм — это внутреннее свойство некоторых языков программирования. Так, для языка Ada механизм параллельных процессов реализуется как *задача.* В Smalltalk есть класс process, которому наследуют все активные объекты. Есть много других языков со встроенными механизмами для параллельного выполнения и синхронизации процессов — Actors, Orient 84/K, ABCL/1, которые предусматривают сходные механизмы параллелизма и синхронизации. Во всех этих языках можно создавать активные объекты, код которых постоянно выполняется параллельно с другими активными объектами.

Во-вторых, можно использовать библиотеку классов, реализующих какую-нибудь разновидность «легкого» параллелизма. Например, библиотека AT&T для C++ содержит классы Shed, Timer, Task и т. д. Ее реализация, естественно, зависит от платформы, хотя интерфейс достаточно хорошо переносим. При этом подходе механизмы параллельного выполнения не встраиваются в язык (и, значит, не влияют на системы без параллельности), но в то же время практически воспринимаются как встроенные.

Наконец, в-третьих, можно создать иллюзию многозадачности с помощью прерываний. Для этого надо кое-что знать об аппаратуре. Например, в нашей реализации класса ActiveTemperatureSensor мы могли бы иметь аппаратный таймер, периодически прерывающий приложение, после чего все датчики измеряли бы температуру и обращались бы, если нужно, к своим функциям вызова.

Как только в систему введен параллелизм, сразу возникает вопрос о том, как синхронизировать отношения активных объектов друг с другом, а также с остальными объектами, действующими последовательно. Например, если два объекта посылают сообщения третьему, должен быть какой-то механизм, гарантирующий, что объект, на который направлено действие, не разрушится при одновременной попытке двух активных объектов изменить его состояние. В этом вопросе соединяются абстракция, инкапсуляция и параллелизм. В параллельных системах недостаточно определить поведение объекта, надо еще принять меры, гарантирующие, что он не будет растерзан на части несколькими независимыми процессами.

Сохраняемость

Любой программный объект существует в памяти и живет во времени. Аткинсон и др. предположили, что есть непрерывное множество продолжительности существования объектов: существуют объекты, которые присутствуют лишь во время вычисления выражения, но есть и такие, как базы данных, которые существуют независимо от программы. Этот спектр сохраняемости объектов охватывает:

- «Промежуточные результаты вычисления выражений.
- Локальные переменные в вызове процедур.
- Собственные переменные (как в ALGOL-60), глобальные переменные и динамически создаваемые данные.
- Данные, сохраняющиеся между сеансами выполнения программы.
- Данные, сохраняемые при переходе на новую версию программы.
- Данные, которые вообще переживают программу» [79].

Традиционно, первыми тремя уровнями занимаются языки программирования, а последними — базы данных. Этот конфликт культур приводит к неожиданным решениям: программисты разрабатывают специальные схемы для сохранения объектов в период между запусками программы, а конструкторы баз данных переиначивают свою технологию под короткоживущие объекты [80].

Унификация принципов параллелизма для объектов позволила создать параллельные языки программирования. Аналогичным образом, введение сохраняемос-ти, как нормальной составной части объектного подхода приводит нас к объектно-ориентированным базам данных (OODB, object-oriented databases). На практике подобные базы данных строятся на основе проверенных временем моделей — последовательных, индексированных, иерархических, сетевых или реляционных, но программист может ввести абстракцию объектно-ориентированного интерфейса, через который запросы к базе данных и другие операции выполняются в терминах

объектов, время жизни которых превосходит время жизни отдельной программы. Как мы увидим в главе 10, эта унификация значительно упрощает разработку отдельных видов приложений, позволяя, в частности, применить единый подход к разным сегментам программы, одни из которых связаны с базами данных, а другие не имеют такой связи.

Языки программирования, как правило, не поддерживают понятия сохраняе-мости; примечательным исключением является Smalltalk, в котором есть протоколы для сохранения объектов на диске и загрузки с диска. Однако, записывать объекты в неструктурированные файлы — это все-таки наивный подход, пригодный только для небольших систем. Как правило, сохраняемость достигается применением (немногочисленных) коммерческих OODB [81]. Другой вариант — создать объектно-ориентированную оболочку для реляционных СУБД; это лучше, в частности, для тех, кто уже вложил средства в реляционную систему. Мы рассмотрим такую ситуацию в главе 10.

Сохраняемость — это не только проблема сохранения *данных.* В OODB имеет смысл сохранять и *классы,* так, чтобы программы могли правильно интерпретировать данные. Это создает большие трудности по мере увеличения объема данных, особенно, если класс объекта вдруг потребовалось изменить.

До сих пор мы говорили о сохранении объектов во времени. В большинстве систем объектам при их создании отводится место в памяти, которое не изменяется и в котором объект находится всю свою жизнь. Однако для распределенных систем желательно обеспечивать возможность перенесения объектов в пространстве, так, чтобы их можно было переносить с машины на машину и даже при необходимости изменять форму представления объекта в памяти. Этими вопросами мы займемся в главе 12.

В заключение определим сохраняемость следующим образом:

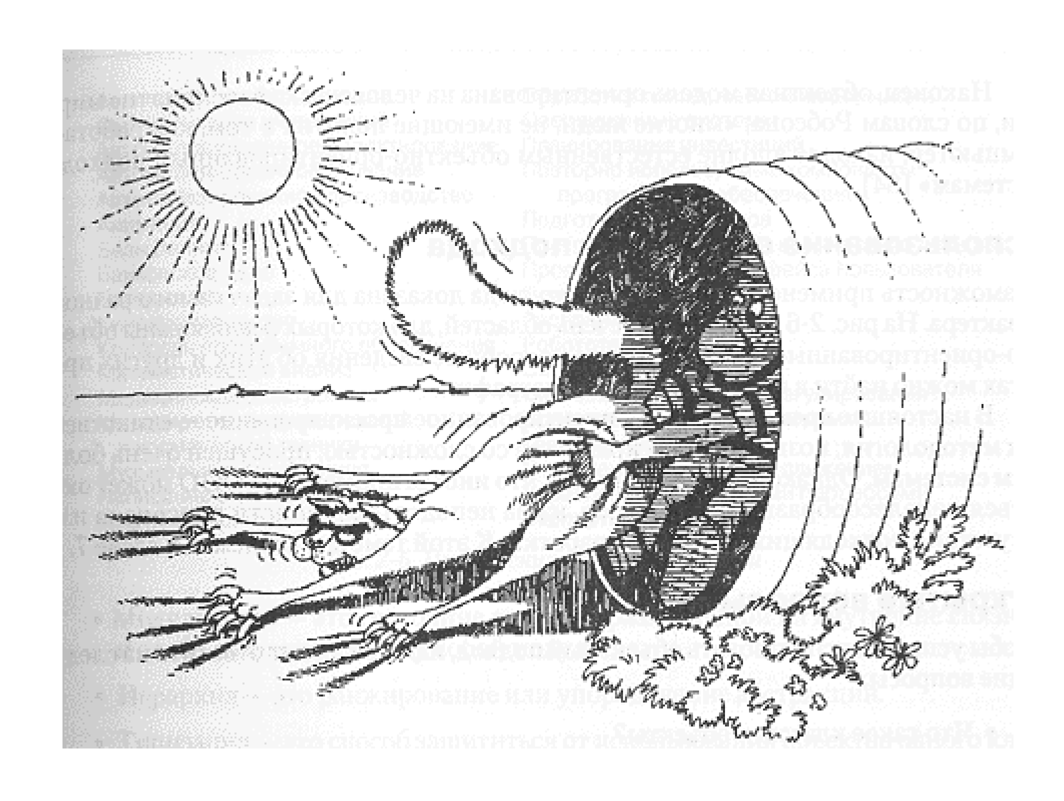
Сохраняемость — способность объекта существовать во времени, переживая породивший его процесс, и (или) в пространстве, перемещаясь из своего первоначального адресного пространства.

2.3. Применение объектной модели

Преимущества объектной модели

Как уже говорилось выше, объектная модель принципиально отличается от моделей, которые связаны с более традиционными методами структурного анализа, проектирования и программирования. Это не означает, что объектная модель требует отказа от всех ранее найденных и испытанных временем методов и приемов. Скорее, она вносит некоторые новые элементы, которые добавляются к предшествующему опыту. Объектный подход обеспечивает ряд существенных удобств, которые другими моделями не предусматривались. Наиболее важно, что объектный подход позволяет создавать системы, которые удовлетворяют пяти признакам хорошо структурированных сложных систем. Согласно нашему опыту, есть еще пять преимуществ, которые дает объектная модель.

Во-первых, объектная модель позволяет в полной мере использовать выразительные возможности объектных и объектно-ориентированных языков программирования. Страуструп отмечает: «Не всегда очевидно, как в полной мере использовать преимущества такого языка, как C++. Существенно повысить эффективность и качество кода можно просто за счет использования C++ в качестве "улучшенного



Сохраняемость поддерживает состояние и класс объекта в пространстве и во времени

C" с элементами абстракции данных. Однако гораздо более значительным достижением является введение иерархии классов в процессе проектирования. Именно это называется OOD и именно здесь преимущества C++ демонстрируются наилучшим образом» [82]. Опыт показал, что при использовании таких языков, как Smalltalk, Object Pascal, C++, CLOS и Ada вне объектной модели, их наиболее сильные стороны либо игнорируются, либо применяются неправильно.

Во-вторых, использование объектного подхода существенно повышает уровень унификации разработки и пригодность для повторного использования не только программ, но и проектов, что в конце концов ведет к созданию среды разработки [83]. Объектно-ориентированные системы часто получаются более компактными, чем их не объектно-ориентированные эквиваленты. А это означает не только уменьшение объема кода программ, но и удешевление проекта за счет использования предыдущих разработок, что дает выигрыш в стоимости и времени.

В-третьих, использование объектной модели приводит к построению систем на основе стабильных промежуточных описаний, что упрощает процесс внесения изменений. Это дает системе возможность развиваться постепенно и не приводит к полной ее переработке даже в случае существенных изменений исходных требований.

В-четвертых, в главе 7 показано, как объектная модель уменьшает риск разработки сложных систем, прежде всего потому, что процесс интеграции растягивается на все время разработки, а не превращается в единовременное событие. Объектный подход состоит из ряда хорошо продуманных этапов проектирования, что также уменьшает степень риска и повышает уверенность в правильности принимаемых решений.

Наконец, объектная модель ориентирована на человеческое восприятие мира, или, по словам Робсона, «многие люди, не имеющие понятия о том, как работает компьютер, находят вполне естественным объектно-ориентированный подход к системам» [84].

Использование объектного подхода

Возможность применения объектного подхода доказана для задач самого разного характера. На рис. 2-6 приведен перечень областей, для которых реализованы объектно-ориентированные системы. Более подробные сведения об этих и других проектах можно найти в приведенной библиографии.

В настоящее время объектно-ориентированное проектирование — единственная методология, позволяющая справиться со сложностью, присущей очень большим системам. Однако, следует заметить, что иногда применение OOD может оказаться нецелесообразным, например, из-за неподготовленности персонала или отсутствия подходящих средств разработки. К этой теме мы вернемся в главе 7.

Открытые вопросы

Чтобы успешно использовать объектный подход, нам предстоит ответить на следующие вопросы:

- Что такое классы и объекты?
- Как идентифицировать классы и объекты в конкретных приложениях?
- Как описать схему объектно-ориентированной системы?
- Как создавать хорошо структурированные объектно-ориентированные системы?
- Как организовать управление процессом разработки согласно OOD? Этим вопросам посвящены следующие пять глав.

Выводы

- Развитие программной индустрии привело к созданию методов объектно-ориентированного анализа, проектирования и программирования, которые служат для программирования «в большом».
- В программировании существует несколько парадигм, ориентированных на процедуры, объекты, логику, правила и ограничения.
- Абстракция определяет существенные характеристики некоторого объекта, которые отличают его от всех других видов объектов и, таким образом, абстракция четко очерчивает концептуальную границу объекта с точки зрения наблюдателя.
- Инкапсуляция — это процесс разделения устройства и поведения объекта;

инкапсуляция служит для того, чтобы изолировать контрактные обязательства абстракции от их реализации.

Авиационное оборудование	Обработка коммерческой информации
Автоматизация учреждении	Операционные системы
Автоматизированное проектирование	Планирование инвестиций
Автоматизированное обучение	Повторно используемые компоненты
Автоматизированное производство	программного обеспечения
Анимация	Подготовка документов
Базы данных	Программные средства космических станций
Банковское дело	Проектирование интерфейса пользователя
Гипермедиа	Проектирование СБИС
Кинопроизводство	Распознавание образов
Контроль программного обеспечения	Робототехника
Математический анализ	Системы телеметрии
Медицинская электроника	Системы управления и регулирования
Моделирование авиационной	Средства разработки программ
и космической техники	Телекоммуникации
Музыкальная композиция	Управление воздушным движением
Написание сценариев	Управление химическими процессами
Нефтяная промышленность	Экспертные системы

Рис. 2-6. Применения объектной модели

- Модульность — это состояние системы, разложенной на внутренне связные и слабо связанные между собой модули.

• Иерархия — это ранжирование или упорядочение абстракций.

• Типизация — это способ защититься от использования объектов одного класса вместо другого, или по крайней мере способ управлять такой подменой.

• Параллелизм — это свойство, отличающее активные объекты от пассивных.

• Сохраняемость — способность объекта существовать во времени и (или) в пространстве.

Дополнительная литература

Концепция объектной модели была впервые введена Джонсом (Jones) [F 1979] и Вильямсом (Williams) [F 1986]. Диссертация Кэя (Кау) [F 1969] дала направление большой части дальнейших работ по объектно-ориентированному программированию.

Шоу (Shaw) [J 1984] дала блестящий обзор механизмов абстракции в языках программирования высокого уровня. Теоретические обоснования абстракции можно найти в работах Лисков и Гуттага (Liskov and Guttag) [H 1986], Гуттага (Guttag) [J 1980] и Хилфингера Hilfinger [J 1982]. Работа Парнаса (Parnas) [F 1979] по скрытию информации была очень плодотворна. Смысл и значение иерархии обсуждается в работе под редакцией Пати (Pattee) [J 1973].

Есть масса литературы по объектно-ориентированному программированию. Карделли и Вегнер (Cardelli and Wegner) [J 1985] и Вегнер (Wegner) X[J 1987] подготовили замечательный обзор объектных и объектно-ориентированных языков. Методические статьи Стефика и Боброва (Stefik and Bobrow) [G 1986], Страуструпа (Stroustrup) [G 1988], Нюгарта(Nygaard)[G 1986] и Грогоно (Grogono) [G 1991] полезны для начала изучения всех вопросов объектно-ориентированного программирования. В книгах Кокса (Cox) [G 1986],Мейера(Meyer)[F 1988],Шмукера(Schmucker) [G 1986] и Кима и Лочовского (Kim and Lochovsky) [F 1989] эти же вопросы рассматриваются более подробно.

Методы объектно-ориентированного проектирования впервые ввел Буч (Booch) [F 1981, 1982, 1986, 1987, 1989]. Методы объектно-ориентированного анализа впервые ввели Шлэер и Меллор (Shiaer and Mellor) [B 1988] и Бэилин (Bailin) [B 1988]. После этого было предложено много методов объектно-ориентированного анализа и проектирования; среди них наиболее значительны были, изложенные в работах Румбаха (Rumbaugh) [F 1991], Коада и Йордона (Coad and Yourdon) [B 1991], Константайна (Constantine) [F 1989], Шлэра и Меллора (Shlaer and Mellor) [B 1992], Мартина и Одела (Martin and Odell) [B 1992], Вассермана (Wasserman) [B 1991], Джекобсона (Jacobson) [F 1992], Рубина и Голдберга (Rubin and Goldberg) [B 1992], Эмбли (Embly) [B 1992], Верфс-Брока (Wirfs-Brock) [F 1990], Голдстейна и Адлера (Goldstein and Adler) [C 1992], Хендерсон-Селлерс (Henderson-Sellers) [F 1992], Файерсмита (Firesmith) [F 1992] и Фьюжина (Fusion) [F 1992].

Разбор конкретных случаев может быть найден в работах Тэйлора (Taylor) [H 1990, C 1992], Берарда (Berard) [H 1993], Лова (Love) [C 1993] и Пинсона с Вейнером (Pinson and Weiner) [C 1990].

Замечательная подборка работ по всем аспектам объектно-ориентированной технологии может быть найдена в трудах Петерсона (Peterson) [G 1987], Шривера и Вегнера (Schriver and Wegner) [G 1987] и Хошафяна и Абнуа (Khoshafian and Abnous) [I 1990]. Труды нескольких ежегодных конференций по объектно-ориентированной технологии — это еще один богатый источник материала. Наиболее интересные форумы — OOPSLA, ECOOP, TOOLS, Object Word и ObjectExpo.

Организации, отвечающие за стандарты по объектной технологии: Object Management Group (OMG) и комитет ANSI X3J7.

Главный источник сведений по C++ — это книга Эллис и Страуструпа (Ellis and Stroustrup) [G 1990]. Другие полезные ссылки: Страуструп (Stroustrup) [G 1991], Липпман (Lippman) I [G 1991] и Коплиен (Coplien) [1992].

Глава 3

Классы и объекты

И инженер, и художник должны хорошо чувствовать материал, с которым они работают. В объектно-ориентированной методологии анализа и создания сложных программных систем основными строительными блоками являются классы и объекты. Выше было дано всего лишь неформальное определение этих двух элементов. В этой главе мы рассмотрим природу классов и объектов, взаимоотношения между ними, а также сообщим несколько полезных правил проектирования хороших абстракций.

3.1. Природа объекта

Что является и что не является объектом?

Способностью к распознанию объектов физического мира человек обладает с самого раннего возраста. Ярко окрашенный мяч привлекает внимание младенца, но, если спрятать мяч, младенец, как правило, не пытается его искать: как только предмет покидает поле зрения, он перестает существовать для младенца. Только в возрасте около года у ребенка появляется представление о предмете: навык, который незаменим для распознавания. Покажите мяч годовалому ребенку и спрячьте его:

скорее всего, ребенок начнет искать спрятанный предмет. Ребенок связывает понятие предмета с постоянством и индивидуальностью формы независимо от действий, выполняемых над этим предметом [1].

В предыдущей главе объект был неформально определен как осязаемая реальность, проявляющая четко выделяемое поведение. С точки зрения восприятия человеком объектом может быть:

- осязаемый и (или) видимый предмет;
- нечто, воспринимаемое мышлением;
- нечто, на что направлена мысль или действие.

Таким образом, мы расширили неформальное определение объекта новой идеей: объект моделирует часть окружающей действительности и таким образом существует во времени и пространстве. Термин *объект* в программном обеспечении впервые был введен в языке Simula и применялся для моделирования реальности [2].

Объектами реального мира не исчерпываются типы объектов, интересные при проектировании программных систем. Другие важные типы объектов вводятся на этапе проектирования, и их взаимодействие друг с другом служит механизмом отображения поведения более высокого уровня [3]. Это приводит нас к более четкому определению, данному Смитом и Токи: "Объект представляет собой конкретный опознаваемый предмет, единицу или сущность (реальную или абстрактную), имеющую четко определенное функциональное назначение в данной предметной области" [4]. В еще более общем плане объект может быть определен как нечто, имеющее четко очерченные границы [5].

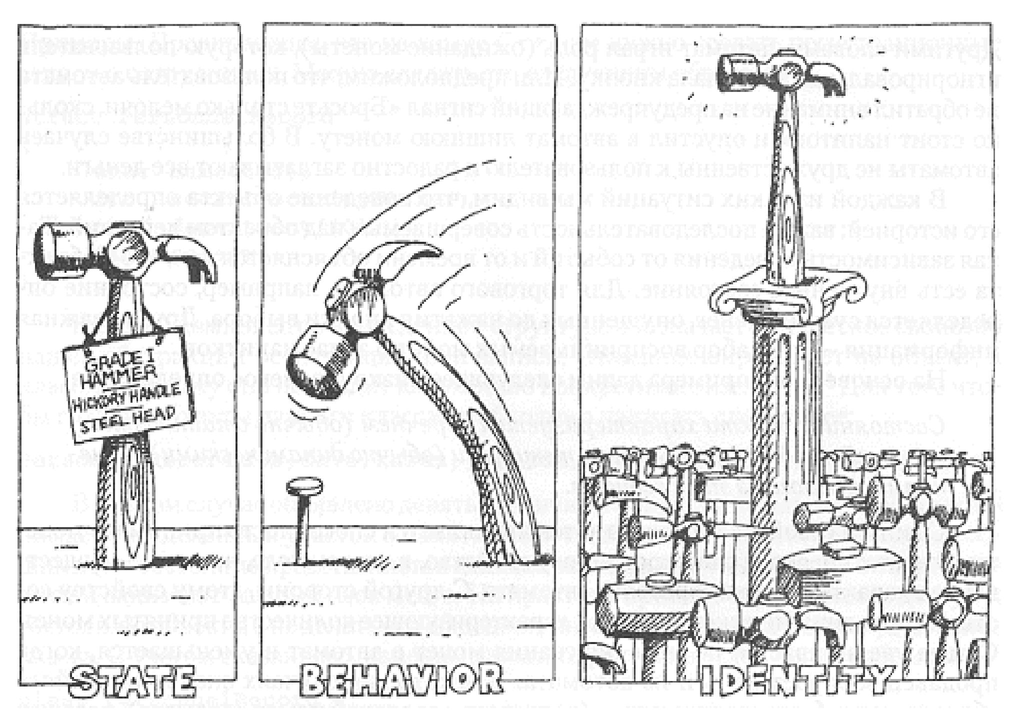
Представим себе завод, на котором создаются композитные материалы для таких различных изделий как, скажем, велосипедные рамы и крылья самолетов. Заводы часто разделяются на цеха: механический, химический, электрический и т. д. Цеха подразделяются на участки, на каждом из которых установлено несколько единиц оборудования: штампы, прессы, станки. На производственных линиях можно увидеть множество емкостей с исходными материалами, из которых с помощью химических процессов создаются блоки композитных материалов. Затем из них делается конечный продукт - рамы или

крылья. Каждый осязаемый предмет может рассматриваться как объект. Токарный станок имеет четко очерченные границы, которые отделяют его от обрабатываемого на этом станке композитного блока; рама велосипеда в свою очередь имеет четкие границы по отношению к участку с оборудованием.

Существуют такие объекты, для которых определены явные концептуальные границы, но сами объекты представляют собой неосязаемые события или процессы. Например, химический процесс на заводе можно трактовать как объект, так как он имеет четкую концептуальную границу, взаимодействует с другими объектами посредством упорядоченного и распределенного во времени набора операций и проявляет хорошо определенное поведение. Рассмотрим систему пространственного проектирования CAD/CAM. Два тела, например, сфера и куб, имеют как правило нерегулярное пересечение. Хотя эта линия пересечения не существует отдельно от сферы и куба, она все же является самостоятельным объектом с четко определенными концептуальными границами.

Объекты могут быть осязаемыми, но иметь размытые физические границы:

реки, туман или толпы людей.[1] Подобно тому, как взявший в руки молоток начина-



Объект имеет состояние, обладает некоторым хорошо определенным поведением и уникальной идентичностью

ет видеть во всем окружающем только гвозди, проектировщик с объектно-ориентированным мышлением начинает воспринимать весь мир в виде объектов. Разумеется, такой взгляд несколько упрощен, так как существуют понятия, явно не являющиеся объектами. К их числу относятся атрибуты, такие, как время, красота, цвет, эмоции (например, любовь или гнев). Однако, потенциально все перечисленное - это свойства, присущие объектам. Можно, например, утверждать, что некоторый человек (объект) любит свою жену (другой объект), или что конкретный кот (еще один объект) - серый.

Полезно понимать, что объект - это нечто, имеющее четко определенные границы, но этого недостаточно, чтобы отделить один объект от

[1] Это верно только на достаточно высоком уровне абстракции. Для человека, идущего через полосу тумана, бессмысленно отличать "мой туман" от "твоего тумана". Однако, рассмотрим карту погоды: полосы тумана в Сан-Франциско и в Лондоне представляют собой совершенно разные объекты.

другого или дать оценку качества абстракции. На основе имеющегося опыта можно дать следующее определение:

Объект обладает состоянием, поведением и идентичностью; структура и поведение схожих объектов определяет общий для них класс; термины "экземпляр класса" и "объект" взаимозаменяемы.

Состояние

Семантика. Рассмотрим торговый автомат, продающий напитки. Поведение такого объекта состоит в том, что после опускания в него монеты и нажатия кнопки автомат выдает выбранный напиток. Что произойдет, если сначала будет нажата кнопка выбора напитка, а потом уже опущена монета? Большинство автоматов при этом просто ничего не сделают, так как пользователь нарушил их основные правила.

Другими словами, автомат играл роль (ожидание монеты), которую пользователь игнорировал, нажав сначала кнопку. Или предположим, что пользователь автомата не обратил внимание на предупреждающий сигнал "Бросьте столько мелочи, сколько стоит напиток" и опустил в автомат лишнюю монету. В большинстве случаев автоматы не дружественны к пользователю и радостно заглатывают все деньги.

В каждой из таких ситуаций мы видим, что поведение объекта определяется его историей: важна последовательность совершаемых над объектом действий. Такая зависимость поведения от событий и от времени объясняется тем, что у объекта есть внутреннее состояние. Для торгового автомата, например, состояние определяется суммой денег, опущенных до нажатия кнопки выбора. Другая важная информация - это набор воспринимаемых монет и запас напитков.

На основе этого примера дадим следующее низкоуровневое определение:

Состояние объекта характеризуется перечнем (обычно статическим) всех свойств данного объекта и текущими (обычно динамическими) значениями каждого из этих свойств.

Одним из свойств торгового автомата является способность принимать монеты. Это статическое (фиксированное) свойство, в том смысле, что оно - существенная характеристика торгового автомата. С другой стороны, этому свойству соответствует динамическое значение, характеризующее количество принятых монет. Сумма увеличивается по мере опускания монет в автомат и уменьшается, когда продавец забирает деньги из автомата. В некоторых случаях значения свойств объекта могут быть статическими (например, заводской номер автомата), поэтому в данном определении использован термин "обычно динамическими".

К числу свойств объекта относятся присущие ему или приобретаемые им характеристики, черты, качества или способности, делающие данный объект самим собой. Например, для лифта характерным является то, что он сконструирован для поездок вверх и вниз, а не горизонтально. Перечень свойств объекта является, как правило, статическим, поскольку эти свойства составляют неизменяемую основу объекта. Мы говорим "как правило", потому что в ряде случаев состав свойств объекта может изменяться. Примером может служить робот с возможностью самообучения. Робот первоначально может рассматривать некоторое препятствие как статическое, а затем обнаруживает, что это дверь, которую можно открыть. В такой ситуации по мере получения новых знаний изменяется создаваемая роботом концептуальная модель мира.

Все свойства имеют некоторые значения. Эти значения могут быть простыми количественными характеристиками, а могут ссылаться на другой объект. Состояние лифта может описываться числом 3, означающим номер этажа, на котором лифт в данный момент находится. Состояние торгового

автомата описывается в терминах других объектов, например, имеющихся в наличии напитков. Конкретные напитки - это самостоятельные объекты, отличные от торгового автомата (их можно пить, а автомат нет, и совершать с ними иные действия).

Таким образом, мы установили различие между объектами и простыми величинами: простые количественные характеристики (например, число 3) являются "постоянными, неизменными и непреходящими", тогда как объекты "существуют во времени, изменяются, имеют внутреннее состояние, преходящи и могут создаваться, уничтожаться и разделяться" [б].

Тот факт, что всякий объект имеет состояние, означает, что всякий объект занимает определенное пространство (физически или в памяти компьютера).

Примеры. Предположим, что на языке C++ нам нужно создать регистрационные записи о сотрудниках. Можно сделать это следующим образом:

```
struct PersonnelRecord {
      char  name[100];
      int   socialSecurityNurober;
      char  department[10];
      float salary;
};
```

Каждый компонент в приведенной структуре обозначает конкретное свойство нашей абстракции регистрационной записи. Описание определяет не объект, а класс, поскольку оно не вводит какой-либо конкретный экземпляр.[2] Для того чтобы создать объекты данного класса, необходимо написать следующее:

```
      PersonnelRecord deb, dave, karen, jim, torn, denise,
kaitlyn, krista, elyse;
```

В данном случае объявлено девять различных объектов, каждый из которых занимает определенный участок в памяти. Хотя свойства этих объектов являются общими (их состояние представляется единообразно), в памяти объекты не пересекаются и занимают каждый свое место. На практике принято ограничивать доступ к состоянию объекта, а не делать его общедоступным, как в предыдущем определении класса. С учетом сказанного, изменим данное определение следующим образом:

```
class PersonnelRecord {
public:
      char* employeeName() const;
      int employeeSocialSecurityNumberf) const;
      char* employeeDepartment() const;
protected:
      char name[100];
      int socialSecurityNumber;
      char department[10];
      float salary;
};
```

Новое определение несколько сложнее предыдущего, но по ряду соображений предпочтительнее.[3] В частности, в новом определении

---

[2] Точнее, это описание определяет структуру в C++, семантика которой соответствует классу, у которого все поля открыты. Таким образом, структуры - это неинкапсулированные абстракции.

[3] К вопросу о стилях: по критериям, которые вводятся в этой главе далее, предложенное определение класса PersonnelRecord - это далеко не шедевр. Мы хотим здесь только показать семантику состояния класса. Иметь в классе функцию, которая возвращает значение char*, часто опасно, так как это нарушает парадигму защиты памяти: если метод отводит себе память, за которую получивший к ней доступ клиент

реализация класса скрыта от других объектов. Если реализация класса будет в дальнейшем изменена, код придется перекомпилировать, но семантически клиенты не будут зависеть от этих изменении (то есть их код сохранится). Кроме того, решается также проблема занимаемой объектом памяти за счет явного определения операций, которые разрешены клиентам над объектами данного класса. В частности, мы даем всем клиентам право узнать имя, код социальной защиты и место работы сотрудника, но только особым клиентам (а именно, подклассам данного класса) разрешено устанавливать значения указанных параметров. Только этим специальным клиентам разрешен доступ к сведениям о заработной плате. Другое достоинство последнего определения связано с возможностью его повторного использования. В следующем разделе мы увидим, что механизм наследования позволяет повторно использовать абстракцию, а затем уточнить и многими способами специализировать ее.

В заключение скажем, что все объекты в системе инкапсулируют некоторое состояние, и все состояние системы инкапсулировано в объекты. Однако, инкапсуляция состояния объекта - это только начало, которого недостаточно, чтобы мы могли охватить полный смысл абстракций, которые мы вводим при разработке. По этой причине нам нужно разобраться, как объекты функционируют.

Поведение

Что такое поведение. Объекты не существуют изолированно, а подвергаются воздействию или сами воздействуют на другие объекты.

Поведение - это то, как объект действует и реагирует; поведение выражается в терминах состояния объекта и передачи сообщений.

Иными словами, поведение объекта - это его наблюдаемая и проверяемая извне деятельность.

Операцией называется определенное воздействие одного объекта на другой с целью вызвать соответствующую реакцию. Например, клиент может активизировать операции append и pop для того, чтобы управлять объектом-очередью (добавить или изъять элемент). Существует также операция length, которая позволяет определить размер очереди, но не может изменить это значение. В чисто объектно-ориентированном языке, таком как Smalltalk, принято говорить о передаче сообщений между объектами. В языках типа C++, в которых четче ощущается процедурное прошлое, мы говорим, что один объект вызывает функцию-член другого. В основном понятие *сообщение* совпадает с понятием *операции* над объектами, хотя механизм передачи различен. Для наших целей эти два термина могут использоваться как синонимы.

В объектно-ориентированных языках операции, выполняемые над данным объектом, называются *методами* и входят в определение класса объекта. В C++ они называются *функциями-членами*. Мы будем использовать эти термины как синонимы.

Передача сообщений - это одна часть уравнения, задающего поведение. Из нашего определения следует, что состояние объекта также влияет на его поведение. Рассмотрим торговый автомат. Мы можем сделать выбор, но поведение автомата будет зависеть от его состояния. Если мы не опустили в него достаточную сумму, скорее всего ничего не произойдет. Если же денег достаточно, автомат выдаст нам желаемое (и тем самым изменит свое состояние). Итак, поведение объекта определяется выполняемыми над

не отвечает, результатом будет замусоривание памяти. В наших системах мы предпочитаем использовать параметризованный класс строк переменной длины, который можно найти в базовой библиотеке классов, вроде той, что описана в главе 9. И еще: классы - это больше чем структуры из C с синтаксисом классов C++; как объясняется в главе 4, классификация требует определенного согласования структуры и поведения.

ним операциями и его состоянием, причем некоторые операции имеют побочное действие: они изменяют состояние. Концепция побочного действия позволяет уточнить наше определение состояния:

Состояние объекта представляет суммарный результат его поведения.

Наиболее интересны те объекты, состояние которых не статично: их состояние изменяется и запрашивается операциями.

Примеры. Опишем на языке C++ класс Queue (очередь):

```
class Queue {
public:
	Queue();
	Queue(const Queue&);
	virtual ~Queue();
	virtual Queue& operator=(const Queue&);
	virtual int operator== (const Queue&) const;
	int operator!=(const Queue&) const;
	virtual void clear();
	virtual void append(const void*);
	virtual void pop();
	virtual void remove(int at);
	virtual int length() const;
	virtual int isEmpty() const;
	virtual const void* front() const;
	virtual int location(const void*);
protected:
…
};
```

В определении класса используется обычная для C идиома ссылки на данные неопределенного типа с помощью void*, благодаря чему в очередь можно вставлять объекты разных классов. Эта техника не безопасна - клиент должен ясно понимать, с каким (какого класса) объектом он имеет дело. Кроме того, при использовании void* очередь не "владеет" объектами, которые в нее помещены. Деструктор ~Queue () уничтожает очередь, но не ее участников. В следующем разделе мы рассмотрим параметризованные типы, которые помогают справляться с такими проблемами.

Так как определение Queue задает класс, а не объект, мы должны объявить экземпляры класса, с которыми могут работать клиенты:

```
Queue a, b, c, d;
```

Мы можем выполнять операции над объектами:

```
a.append(&deb);
a.append(&karen);
a. append (&denise);
b = a;
a.pop();
```

Теперь очередь a содержит двух сотрудников (первой стоит karen), а очередь b - троих (первой стоит deb). Таким образом, очереди имеют определенное состояние, которое влияет на их будущее поведение - например, одну очередь можно безопасно продвинуть (pop) еще два раза, а вторую - три.

Операции. Операция - это услуга, которую класс может предоставить своим клиентам. На практике типичный клиент совершает над объектами

операции пяти видов.[4] Ниже приведены три наиболее распространенные операции:

• Модификатор Операция, которая изменяет состояние объекта.

• Селектор Операция, считывающая состояние объекта, но не меняющая состояния.

• Итератор Операция, позволяющая организовать доступ ко всем частям объекта в строго определенной последовательности.

Поскольку логика этих операций весьма различна, полезно выбрать такой стиль программирования, который учитывает эти различия в коде программы. В нашей спецификации класса Queue мы вначале перечислили все модификаторы (функции-члены без спецификаторов const - clear, append, pop, remove), а потом все селекторы (функции со спецификаторами const - length, `isEmpty`, front и `location`). Позднее в главе 9, следуя нашему стилю, мы определим отдельный класс, который действует как агент, отвечающий за итеративный просмотр очередей.

Две операции являются универсальными; они обеспечивают инфраструктуру, необходимую для создания и уничтожения экземпляров класса:

• Конструктор Операция создания объекта и/или его инициализации.

• Деструктор Операция, освобождающая состояние объекта и/или разрушающая сам объект.

В языке C++ конструктор и деструктор составляют часть описания класса, тогда как в Smalltalk и CLOS эти операторы определены в протоколе метакласса (то есть класса класса).

В чисто объектно-ориентированных языках, таких как Smalltalk, операции могут быть только методами, так как процедуры и функции вне классов в этом языке определять не допускается. Напротив, в языках Object Pascal, C++, CLOS и Ada допускается описывать операции как независимые от объектов подпрограммы. В C++ они называются функциями-нечленами; мы же будем здесь называть их свободными подпрограммами. Свободные подпрограммы - это процедуры и функции, которые выполняют роль операций высокого уровня над объектом или объектами одного или разных классов. Свободные процедуры группируются в соответствии с классами, для которых они создаются. Это дает основание называть такие пакеты процедур утилитами класса. Например, для определенного выше класса Queue можно написать следующую свободную процедуру:

```
void copyUntilFound(Queue& from. Queue& to, void* item)
{
      while ((!from.isEmpty()) && (from.front() != item))
      {
            to.append(from.front()) ;
            from.pop();
      }
}
```

Смысл в том, что содержимое одной очереди переходит в другую до тех пор, пока в голове первой очереди не окажется заданный объект. Это операция высокого уровня; она строится на операциях-примитивах класса Queue.

В C++ (и Smalltalk) принято собирать все логически связанные свободные подпрограммы и объявлять их частью некоторого класса, не имеющего состояния. Все такие функции будут статическими.

---

[4] Липпман предложил несколько иную классификацию: функции управления, функции реализации, вспомогательные функции (все виды модификаторов) и функции доступа (эквивалентные селекторам) [7].

Таким образом, можно утверждать, что все методы - операции, но не все операции - методы: некоторые из них представляют собой свободные подпрограммы. Мы склонны использовать только методы, хотя, как будет показано в следующем разделе, иногда трудно удержаться от искушения, особенно если операция по своей природе выполняется над несколькими объектами разных классов и нет никаких причин объявить ее операцией именно одного класса, а не другого.

Роли и ответственности. Совокупность всех методов и свободных процедур, относящихся к конкретному объекту, образует протокол этого объекта. Протокол, таким образом, определяет поведение объекта, охватывающее все его статические и динамические аспекты. В самых нетривиальных абстракциях полезно подразделять протокол на частные аспекты поведения, которые мы будет называть ролями. Адаме говорит, что роль - это маска, которую носит объект [8]; она определяет контракт абстракции с ее клиентами.

Объединяя наши определения состояния и поведения объекта, Вирфс-Брок вводит понятие ответственности. "Ответственности объекта имеют две стороны - знания, которые объект поддерживает, и действия, которые объект может исполнить. Они выражают смысл его предназначения и место в системе. Ответственность понимается как совокупность всех услуг и всех контрактных обязательств объекта" [9]. Таким образом можно сказать, что состояние и поведение объекта определяют исполняемые им роли, а те, в свою очередь, необходимы для выполнения ответственности данной абстракции.

Действительно большинство интересных объектов исполняют в своей жизни разные роли, например [10]:

• Банковский счет может быть в хорошем или плохом состоянии (две роли), и от этой роли зависит, что произойдет при попытке снятия с него денег.

• Для фондового брокера пакет акций - это товар, который можно покупать или продавать, а для юриста это знак обладания определенными правами.

• В течении дня одна и та же персона может играть роль матери, врача, садовника и кинокритика.

Роли банковского счета являются динамическими и взаимоисключающими. Роли пакета акций слегка перекрываются, но каждая из них зависит от того, что клиент с ними делает. В случае персоны роли динамически изменяются каждую минуту.

Как мы увидим в главах 4 и 6, мы часто начинаем наш анализ с перечисления разных ролей, которые может играть объект. Во время проектирования мы выделяем эти роли, вводя конкретные операции, выполняющие ответственности каж-дой роли.

Объекты как автоматы. Наличие внутреннего состояния объектов означает, что порядок выполнения операций имеет существенное значение. Это наводит на мысль представить объект в качестве маленькой независимой машины [II]. Действительно, для ряда объектов такой временной порядок настолько существен, что наилучшим способом их формального описания будет конечный автомат. В главе 5 мы введем обозначения для описания иерархических конечных автоматов, которые можно использовать для выражения соответствующей семантики.

Продолжая аналогию с машинами, можно сказать, что объекты могут быть активными и пассивными. Активный объект имеет свой поток управления, а пассивный - нет. Активный объект в общем случае автономен, то есть он может проявлять свое поведение без воздействия со стороны других объектов. Пассивный объект, напротив, может изменять свое состояние только под воздействием других объектов. Таким образом, активные объекты системы - источники управляющих воздействий. Если система имеет

несколько потоков управления, то и активных объектов может быть несколько. В последовательных системах обычно в каждый момент времени существует только один активный объект, например, главное окно, диспетчер которого ловит и обрабатывает все сообщения. В таком случае остальные объекты пассивны: их поведение проявляется, когда к ним обращается активный объект. В других видах последовательных архитектур (системы обработки транзакций) нет явного центра активности, и управление распределено среди пассивных объектов системы.

Идентичность

Семантика. Хошафян и Коуплэнд предложили следующее определение:

"Идентичность - это такое свойство объекта, которое отличает его от всех других объектов" [12].

Они отмечают, что "в большинстве языков программирования и управления базами данных для различения временных объектов их именуют, тем самым путая адресуемость и идентичность. Большинство баз данных различают постоянные объекты по ключевому атрибуту, тем самым смешивая идентичность и значение данных". Источником множества ошибок в объектно-ориентированном программировании является неумение отличать имя объекта от самого объекта.

Примеры. Начнем с определения точки на плоскости.

```
struct Point {
	int x;
	int y;
	Point() : x(0), y(0) {}
	Point(int xValue, int yValue) : x(xValue), y(yValue) {}
};
```

Мы определили point как структуру, а не как полноценный класс. Правило, на основании которого мы так поступили, очень просто. Если абстракция представляет собой собрание других объектов без какого-либо собственного поведения, мы делаем ее структурой. Однако, когда наша абстракция подразумевает более сложное поведение, чем простой доступ к полям структуры, то нужно определять класс. В данном случае абстракция point - это просто пара координат (x,y). Для удобства предусмотрено два конструктора: один инициализирует точку нулевыми значениями координат, а другой - некоторыми заданными значениями.

Теперь определим экранный объект (Displayltem). Это абстракция довольно обычна для систем с графическим интерфейсом (GUI) - она является базовым классом для всех объектов, которые можно отображать в окне. Мы хотим сделать его чем-то большим, чем просто совокупностью точек. Надо, чтобы клиенты могли рисовать, выбирать объекты и перемещать их по экрану, а также запрашивать их положение и состояние. Мы записываем нашу абстракцию в виде следующего объявления на C++:

```
class Displayltem {
public:
	Displayltem();
	Displayltem(const Point& location);
	virtual ~Displayltem();
	virtual void draw();
	virtual void erase();
	virtual void select();
	virtual void unselect();
	virtual void move(const Point& location);
	int isSelected() const;
	Point location() const;
	int isUnder(const Point& location) const;
```

```
protected:
…
};
```

В этом объявлении мы намеренно опустили конструкторы, а также операторы для копирования, присваивания и проверки на равенство. Их мы оставим до следующего раздела.

Мы ожидаем, что у этого класса будет много наследников, поэтому деструктор и все модификаторы объявлены виртуальными. В особенности это относится к draw. Напротив, селекторы скорее всего не будут переопределяться в подклассах. Заметьте, что один из них, isUnder, должен вычислять, накрывает ли объект данную точку, а не просто возвращать значение какого-то свойства.

Объявим экземпляры указанных классов:

```
DisplayItem item1;
Displayltem* item2 = new DisplayItem(Point(75, 75));
Displayltem* item3 = new Displayltem(Point(100, 100));
Displayltem* item4 = 0;
```

Рис. 3-1 а показывает, что при выполнении этих операторов возникают четыре имени и три разных объекта. Конкретно, в памяти будут отведены четыре места под имена **item1**, **item2**, **item3**, **item4**. При этом **iteml** будет именем объекта класса Displayltem, а три других будут указателями. Кроме того, лишь **item2** и **item3** будут на самом деле указывать на объекты класса Displayltem. У объектов, на которые указывают **item2** и **item3**, к тому же нет имен, хотя на них можно ссылаться "разыменовывая" соответствующие указатели: например, ***item2**. Поэтому мы можем сказать, что **item2** указывает на отдельный объект класса Displayltem, на

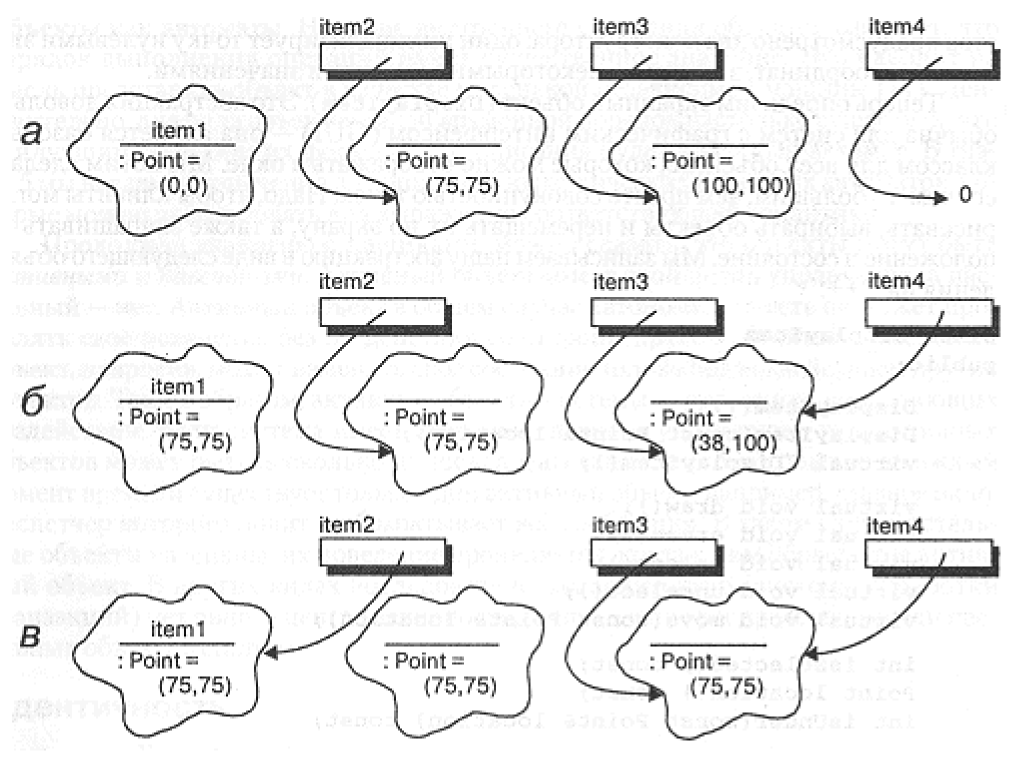


Рис. 3-1. Идентичность объектов

имя которого мы можем косвенно ссылаться через ***item2**. Уникальная идентичность (но не обязательно имя) каждого объекта сохраняется на все время его существования, даже если его внутреннее состояние изменилось. Эта ситуация напоминает парадокс Зенона о реке: может ли река быть той же самый, если в ней каждый день течет разная вода?

Рассмотрим результат выполнения следующих операторов (рис. 3-1 б):

```
item1.move(item2->location());
item4 = item3;
item4->move(Point(38, 100));
```

Объект **iteml** и объект, на который указывает **item2**, теперь относятся к одной и той же точке экрана. Указатель **item4** стал указывать на тот же объект, что и **item3**. Кстати, заметьте разницу между выражениями "объект **item2**" и "объект, на который указывает **item2**". Второе выражение более точно, хотя для краткости мы часто будем использовать их как синонимы.

Хотя объект **item1** и объект, на который указывает **item2**, имеют одинаковое состояние, они остаются разными объектами. Кроме того, мы изменили состояние объекта ***item3**, использовав его новое косвенное имя **item4**. Эта ситуация, которую мы называем структурной зависимостью, подразумевая под этим ситуацию, когда объект именуется более чем одним способом несколькими синонимичными именами. Структурная зависимость порождает в объектно-ориентированном программировании много проблем. Трудность распознания побочных эффектов при действиях с синонимичными

объектами часто приводит к "утечкам памяти", неправильному доступу к памяти, и, хуже того, непрогнозируемому изменению состояния. Например, если мы уничтожим объект через указатель **item3**, то значение указателя **item4** окажется бессмысленным; эта ситуация называется повисшей ссылкой. На рис. 3-1в иллюстрируется результат выполнения следующих действий:

```
item2 = &item1;
item4->move(item2->location());
```

В первой строке создается синоним: **item2** указывает на тот же объект, что и **iteml**. Во второй доступ к состоянию **iteml** получен через этот новый синоним. К сожалению, при этом произошла утечка памяти, - объект, на который первоначально указывала ссылка **item2**, никак не именуется ни прямо, ни косвенно, и его идентичность потеряна. В Smalltalk и CLOS память, отведенная под объекты, будет вновь возвращена системе сборщиком мусора. В языках типа C++ такая память не освобождается, пока не завершится программа, создавшая объект. Такие утечки памяти могут вызвать и просто неудобство, и крупные сбои, особенно, если программа должна непрерывно работать длительное время.[5]

Копирование, присваивание и равенство. Структурная зависимость имеет место, когда объект имеет несколько имен. В наиболее интересных приложениях объектно-ориентированного подхода использование синонимов просто неизбежно. Например, рассмотрим следующие две функции:

```
void highLight(DisplayItem& i);
void drag(DisplayItem i); // Опасно
```

Если вызвать первую функцию с параметром **iteml**, будет создан псевдоним:

формальный параметр **i** означает указатель на фактический параметр, и следовательно **iteml** и **i** именуют один и тот же объект во время выполнения функции. При вызове второй функции с аргументом **iteml** ей передается новый объект, являющийся копией **iteml**: **i** обозначает совершенно другой объект, хотя и с тем же состоянием, что и **iteml**. В C++ различается передача параметров по ссылке и по значению. Надо следить за этим, иначе можно нечаянно изменить копию объекта, желая изменить сам объект.[6] Как мы увидим в следующем разделе, передача объектов по ссылке в C++ необходима для программирования полиморфного поведения. В общем случае, передача объектов по ссылке крайне желательна для достаточно сложных объектов, поскольку при этом копируется ссылка, а не состояние, и следовательно, достигается большая эффективность (за исключением тех случаев, когда передаваемое значение очень простое).

В некоторых обстоятельствах, однако, подразумевается именно копирование. В языках типа C++ семантику копирования можно контролировать. В частности, мы можем ввести копирующий конструктор в определение класса, как в следующем фрагменте кода, который можно было бы включить в описание класса **Displayitem**:

```
Displayltem(const Displayltem&);
```

В C++ копирующий конструктор может быть вызван явно (как часть описания объекта) или неявно (с передачей объекта по значению). Отсутствие этого специального конструктора вызывает копирующий конструктор,

[5] Представьте себе утечку памяти в программе управления спутником или сердечным стимулятором. Перезапуск компьютера на спутнике в нескольких миллионах километров от Земли очень неудобен. Аналогично, непредсказуемая сборка мусора в программе, управляющей стимулятором, может оказаться смертельным для пациента. В таких случаях разработчики систем реального времени предпочитают воздерживаться от динамического распределения памяти.

[6] В Smalltalk семантика передачи объектов методам в качестве аргументов является по своему духу эквивалентом передачи параметра по ссылке в C++.

действующий по умолчанию, который копирует объект поэлементно. Однако, когда объект содержит ссылки или указатели на другие объекты, такая операция приводит к созданию синонимов указателей на объекты, что делает поэлементное копирование опасным. Мы предлагаем эмпирическое правило: разрешать неявное размножение путем копирования только для объектов, содержащих исключительно примитивные значения, и делать его явным для более сложных объектов.

Это правило поясняет то, что некоторые языки называют "поверхностным" и "глубоким" копированием. Чтобы копировать объект, в языке Smalltalk введены методы **shallowCopy** (метод копирует только объект, а состояние является разделяемым) и **deepCopy** (метод копирует объект и состояние, если нужно - рекурсивно). Переопределяя эти методы для классов с агрегацией, можно добиваться эффекта "глубокого" копирования для одних частей объекта, и "поверхностного" копирования остальных частей.

Присваивание - это, вообще говоря, копирование. В C++ его смысл тоже можно изменять. Например, мы могли бы добавить в определение класса **DisplayItem** следующую строку:

```
virtual DisplayItem& operator=(const Displayltem&);
```

Этот оператор намеренно сделан виртуальным, так как ожидается, что подклассы будут его переопределять. Как и в случае копирующего конструктора, копирование можно сделать "глубоким" и "поверхностным". Если оператор присваивания не переопределен явно, то по умолчанию объект копируется поэлементно.

С вопросом присваивания тесно связан вопрос равенства. Хотя вопрос кажется простым, равенство можно понимать двумя способами. Во-первых, два имени могут обозначать один и тот же объект. Во-вторых, это может быть равенство состояний у двух разных объектов. В примере на рис. 3-1в оба варианта тождественности будут справедливы для **iteml** и **item2**. Однако для **item2** и **item3** истинным будет только второй вариант.

В C++ нет предопределенного оператора равенства, поэтому мы должны определить равенство и неравенство, объявив эти операторы при описании:

```
virtual int operator=(const Displayltem&) const;
int operator!=(const Displayltem&) const;
```

Мы предлагаем описывать оператор равенства как виртуальный (так как ожидаем, что подклассы могут переопределять его поведение), и описывать оператор неравенства как невиртуальный (так как хотим, чтобы он всегда был логическим отрицанием равенства: подклассам не следует переопределять это).

Аналогичным образом мы можем создавать операторы сравнения объектов типа >= и <=.

Время жизни объектов. Началом времени существования любого объекта является момент его создания (отведение участка памяти), а окончанием - возвращение отведенного участка памяти системе.

Объекты создаются явно или неявно. Есть два способа создать их явно. Во-первых, это можно сделать при объявлении (как это было с **item1**): тогда объект размещается в стеке. Во-вторых, как в случае **item3**, можно разместить объект, то есть выделить ему память из "кучи". В C++ в любом случае при этом вызывается конструктор, который выделяет известное ему количество правильно инициализированной памяти под объект. В Smalltalk этим занимаются метаклассы, о семантике которых мы поговорим позже.

Часто объекты создаются неявно. Так, передача параметра по значению в C++ создает в стеке временную копию объекта. Более того, создание объектов транзи-тивно: создание объекта тянет за собой создание других объектов, входящих в него. Переопределение семантики копирующего

конструктора и оператора присваивания в C++ разрешает явное управление тем, когда части объекта создаются и уничтожаются. К тому же в C++ можно переопределять и оператор **new**, тем самым изменяя политику управления памятью в "куче" для отдельных классов.

В Smalltalk и некоторых других языках при потере последней ссылки на объект его забирает сборщик мусора. В языках без сборки мусора, типа C++, объекты, созданные в стеке, уничтожаются при выходе из блока, в котором они были определены, но объекты, созданные в "куче" оператором **new**, продолжают существовать и занимать место в памяти: их необходимо явно уничтожать оператором **delete**. Если объект "забыть", не уничтожить, это вызовет, как уже было сказано выше, утечку памяти. Если же объект попробуют уничтожить повторно (например, через другой указатель), последствием будет сообщение о нарушении памяти или полный крах системы.

При явном или неявном уничтожении объекта в C++ вызывается соответствующий деструктор. Его задача не только освободить память, но и решить, что делать с другими ресурсами, например, с открытыми файлами.[7]

Уничтожение долгоживущих объектов имеет несколько другую семантику. Как говорилось в предыдущей главе, некоторые объекты могут быть долгоживу-щими; под этим понимается, что их время жизни может выходить за время жизни породивших их программ. Обычно такие объекты являются частью некой долговременной объектной структуры, поэтому вопросы их жизни и смерти относятся скорее к политике соответствующей объектно-ориентированной базы данных. В таких системах для обеспечения долгой жизни наиболее принят подход на основе постоянных "подмешиваемых классов". Все объекты, которым мы хотим обеспечить долгую жизнь, должны наследовать от этих классов.

3.2. Отношения между объектами

Типы отношений

Сами по себе объекты не представляют никакого интереса: только в процессе взаимодействия объектов реализуется система. По выражению Ингалса: "Вместо процессора, беззастенчиво перемалывающего структуры данных, мы получаем сообщество хорошо воспитанных объектов, которые вежливо просят друг друга об услугах" [13]. Самолет, по определению, "совокупность элементов, каждый из которых по своей природе стремится упасть на землю, но за счет совместных непрерывных усилий преодолевающих эту тенденцию" [14]. Он летит только благодаря согласованным усилиям своих компонентов.

Отношения двух любых объектов основываются на предположениях, которыми один обладает относительно другого: об операциях, которые можно выполнять, и об ожидаемом поведении. Особый интерес для объектно-ориентированного анализа и проектирования представляют два типа иерархических соотношений объектов:

- связи
- агрегация.

Зайдевиц и Старк назвали эти два типа отношений отношениями старшинства и "родитель/потомок" соответственно [15].

Связи

---

[7] Деструкторы не освобождают автоматически память, размещенную оператором **new**, программисты должны явно освободить ее.

Семантика. Мы заимствуем понятие связи у Румбаха, который определяет его как "физическое или концептуальное соединение между объектами" [16]. Объект сотрудничает с другими объектами через связи, соединяющие его с ними. Другими словами, связь - это специфическое сопоставление, через которое клиент запрашивает услугу у объекта-сервера или через которое один объект находит путь к другому.

На рис. 3-2 показано несколько разных связей. Они отмечены линиями и означают как бы пути прохождения сообщений. Сами сообщения показаны стрелками (соответственно их направлению) и помечены именем операции. На рисунке объект **aController** связан с двумя объектами класса **Displayltem** (объекты а и ь). В свою очередь, оба, вероятно, связаны с **aView**, но нам была интересна только одна из этих связей. Только вдоль связи один объект может послать сообщение другому.

Связь между объектами и передача сообщений обычно односторонняя (как на рисунке; хотя технически она может быть и взаимной). Как мы увидим в главе 5, подобное разделение прав и обязанностей типично для хорошо структурированных объектных систем.[8] Заметьте также, что хотя передаваемое сообщение инициализировано клиентом (в данном случае **aController**), данные передаются в обоих направлениях. Например, когда **aController** вызывает операцию move для пересылки данных объекту а, данные передаются от клиента серверу, но при выполнении операции **isUnder** над объектом **b**, результат передается от сервера клиенту.

Участвуя в связи, объект может выполнять одну из следующих трех ролей:

• Актер[9] Объект может воздействовать на другие объекты, но сам никогда не подвергается воздействию других объектов; в определенном смысле это соответствует понятию активный объект

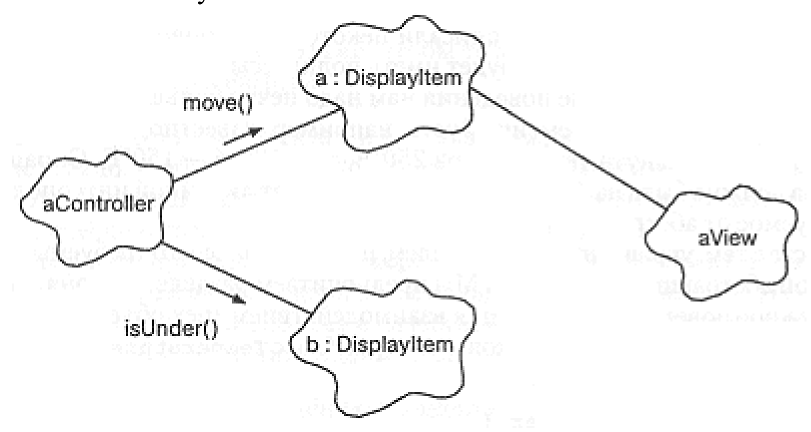


Рис. 3-2. Связи

• Сервер Объект может только подвергаться воздействию со стороны других объектов, но он никогда не выступает в роли воздействующего объекта

• Агент Такой объект может выступать как в активной, так и в пассивной роли; как правило, объект-агент создается для выполнения операций в интересах какого-либо объекта-актера или агента.

На рис. 3-2 объект **aController** выступает как актер, объект а - как агент и объект **aView** - как сервер.

---

[8] На самом деле организация объектов, показанная на рис. 3-2, встречается настолько часто, что ее можно считать типовым проектным решением. В Smalltalk аналогичный механизм известен как MVC, model/view/controller (модель/представление/контроллер). Как мы увидим в следующей главе, хорошо структурированные системы имеют много таких опознаваемых типовых решений.

[9] Actor - это деятель, исполнитель. А исполнитель ролей, это и есть актер. - Примеч.ред.

Пример. Во многих промышленных процессах требуется непрерывное изменение температуры. Необходимо поднять температуру до заданного значения, выдержать заданное время и понизить до нормы. Профиль изменения температуры у разных процессов разный; зеркало телескопа надо охлаждать очень медленно, а закаляемую сталь очень быстро.

Абстракция нагрева имеет достаточно четкое поведение, что дает нам право на описание такого класса. Сначала определим тип, значение которого задает прошедшее время в минутах.

```
// Число прошедших минут
typedef unsigned int Minute;
```

Теперь опишем сам класс **TemperatureRamp**, который по смыслу задает функцию времени от температуры:

```
class TemperatureRamp {
public:
	TemperatureRamp() ;
	virtual ~TenperatureRamp() ;
	virtual void clear();
	virtual void bind (Temperature, Minute);
	Temperature TemperatureAt(Minute);
protected:
…
	};
```

Выдерживая наш стиль, мы описали некоторые из операций виртуальными, гак как ожидаем, что этот класс будет иметь подклассы.

На самом деле в смысле поведения нам надо нечто большее, чем просто зависимость температуры от времени. Пусть, например, известно, что на 60-й минуте должна быть достигнута температура 250°F, а на 180-й - 150°F. Спрашивается, какой она должна быть на 120-й минуте? Это требует линейной интерполяции, так что требуемое от абстракции поведение усложняется.

Вместе с тем, управления нагревателем, поддерживающего требуемый профиль, мы от этой абстракции не требуем. Мы предпочитаем разделение понятий, при котором нужное поведение достигается взаимодействием трех объектов: экземпляра **TemperatureRamp**, нагревателя и контроллера. Класс **TemperatureController** можно определить так:

```
class TemperatureController {
public:
	TemperatureController(Location);
	~TemperatureController();
	void process(const TemperatureRamp&) ;
	Minute schedule(const TemperatureRamp&) const;
private:
…
	};
```

Тип **Location** был определен в главе 2. Заметьте, что мы не ожидаем наследования от этого класса и поэтому не объявляем в нем никаких виртуальных функций.

Операция **process** обеспечивает основное поведение этой абстракции; ее назначение - передать график изменения температуры нагревателю, установленному в конкретном месте. Например, объявим:

```
TemperatureRamp growingRamp;
TemperatureController rampController(7) ;
```

Теперь зададим пару точек и загрузим план в контроллер:

```
growingRamp. bind (250, 60);
growingRamp.bind(150, 180);
rampController.process(growingRamp) ;
```

В этом примере **rampController** - агент, ответственный за выполнение температурного плана, он использует объект **growingRamp** как сервер. Эта связь проявляется хотя бы в том, что **rampController** явно получает **growingRamp** в качестве параметра одной из своих операций.

Одно замечание по поводу нашего стиля. На первый взгляд может показаться, что наши абстракции - лишь объектные оболочки для элементов, полученных в результате обычной функциональной декомпозиции. Пример операции **schedule** показывает, что это не так. Объекты класса **TemperatureController** имеют достаточно интеллекта, чтобы определять расписание для конкретных профилей, и мы включаем эту операцию как дополнительное поведение нашей абстракции. В некоторых энергоемких технологиях (например, плавка металлов) можно существенно выиграть, если учитывать остывание установки и тепло, остающееся после предыдущей плавки. Поскольку существует операция **schedule**, клиент может запросить объект **TemperatureController**, чтобы тот рекомендовал оптимальный момент запуска следующего нагрева.

Видимость. Пусть есть два объекта А и в и связь между ними. Чтобы А мог послать сообщение в, надо, чтобы в был в каком-то смысле видим для А. Мы можем не заботиться об этом на стадии анализа, но когда дело доходит до реализации системы, мы должны обеспечить видимость связанных объектов.

В предыдущем примере объект **rampController** видит объект **growingRamp**, поскольку оба они объявлены в одной области видимости и потому, что **growingRamp** передается объекту **rampController** в качестве параметра. В принципе есть следующие четыре способа обеспечить видимость.

• Сервер глобален по отношению к клиенту.

• Сервер (или указатель на него) передан клиенту в качестве параметра операции.

• Сервер является частью клиента.

• Сервер локально порождается клиентом в ходе выполнения какой-либо операции.

Какой именно из этих способов выбрать - зависит от тактики проектирования.

Синхронизация. Когда один объект посылает по связи сообщение другому, связанному с ним, они, как говорят, синхронизируются. В строго последовательном приложении синхронизация объектов и состоит в запуске метода (см. врезку ниже). Однако в многопоточной системе объекты требуют более изощренной схемы передачи сообщений, чтобы разрешить проблемы взаимного исключения, типичные для параллельных систем. Активные объекты сами по себе выполняются как потоки, поэтому присутствие других активных объектов на них обычно не влияет. Если же активный объект имеет связь с пассивным, возможны следующие три подхода к синхронизации:

• Последовательный - семантика пассивного объекта обеспечивается в присутствии только одного активного процесса.

• Защищенный - семантика пассивного объекта обеспечивается в присутствии многих потоков управления, но активные клиенты должны договориться и обеспечить взаимное исключение.

• Синхронный - семантика пассивного объекта обеспечивается в присутствии многих потоков управления; взаимное исключение обеспечивает сервер.

Все объекты, описанные в этой главе, были последовательными. В главе 9 мы рассмотрим остальные варианты более подробно.

Агрегация

Семантика. В то время, как связи обозначают равноправные или "клиент-серверные" отношения между объектами, агрегация описывает отношения целого и части, приводящие к соответствующей иерархии объектов, причем, идя от целого (агрегата), мы можем придти к его частям (атрибутам). В этом смысле агрегация - специализированный частный случай ассоциации. На рис. 3-3 объект `rampController`

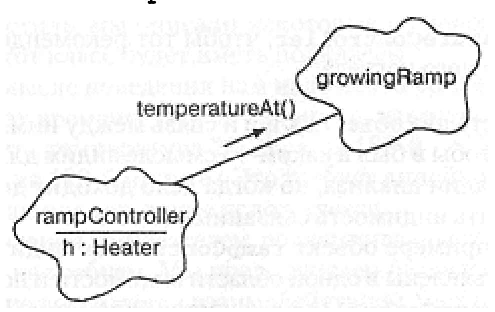


Рис. 3-3. Агрегация

имеет связь с объектом **`growingRamp`** и атрибут h класса **`Heater`** (нагреватель). В данном случае **`rampController`** - целое, а h- его часть. Другими словами, h - часть состояния **`rampController`**. Исходя из **`rampController`**, можно найти соответствующий нагреватель. Однако по h нельзя найти содержащий его объект (называемый также его контейнером), если только сведения о нем не являются случайно частью состояния h.

Агрегация может означать физическое вхождение одного объекта в другой, но не обязательно. Самолет состоит из крыльев, двигателей, шасси и прочих частей. С другой стороны, отношения акционера с его акциями - это агрегация, которая не предусматривает физического включения. Акционер монопольно владеет своими акциями, но они в него не входят физически. Это, несомненно, отношение агрегации, но скорее концептуальное, чем физическое по своей природе.

Выбирая одно из двух - связь или агрегацию - надо иметь в виду следующее. Агрегация иногда предпочтительнее, поскольку позволяет скрыть части в целом. Иногда наоборот предпочтительнее связи, поскольку они слабее и менее ограничительны. Принимая решение, надо взвесить все.

Объект, являющийся атрибутом другого объекта (агрегата), имеет связь со своим агрегатом. Через эту связь агрегат может посылать ему сообщения.

`Пример.` Добавим в спецификацию класса **`TemperatureController`** описание нагревателя:

```
Heater h;
```

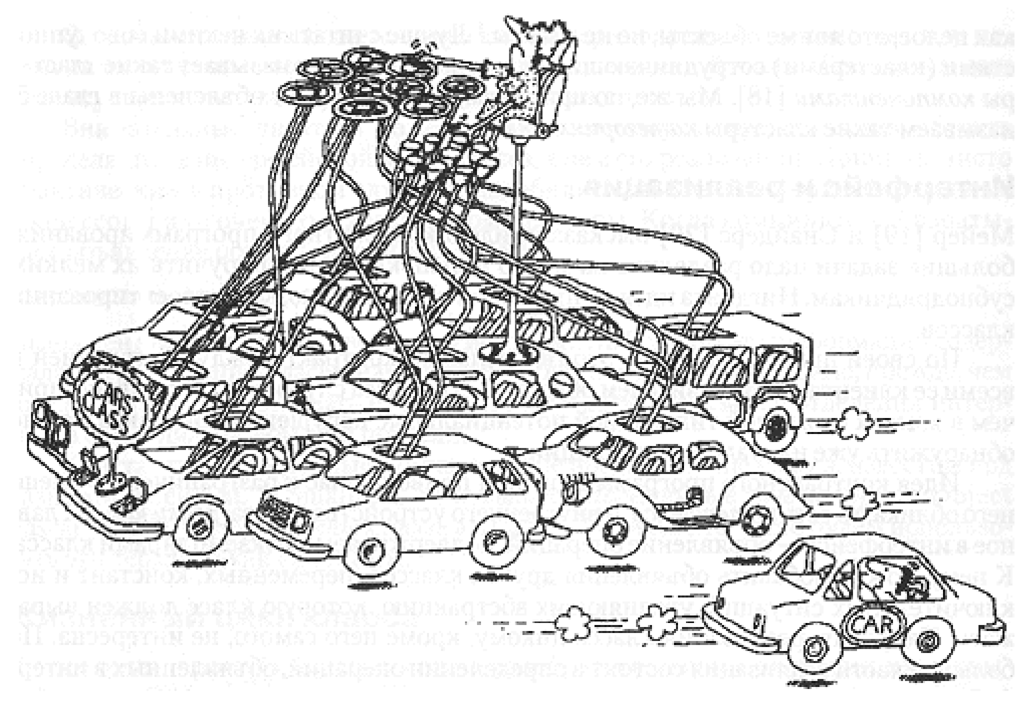
После этого каждый объект **`TemperatureController`** будет иметь свой нагреватель. В соответствии с нашим определением класса Heater в предыдущей главе мы должны инициализировать нагреватель при создании нового контроллера, так как сам этот класс не предусматривает конструктора по умолчанию. Мы могли бы определить конструктор класса **`TemperatureController`** следующим образом:

```
TemperatureController::TemperatureController(Location 1) :
h(1) {}
```

3.3. Природа классов

Что такое класс?

Понятия класса и объекта настолько тесно связаны, что невозможно говорить об объекте безотносительно к его классу. Однако существует важное различие этих



Класс представляет набор объектов, которые обладают общей структурой и одинаковым поведением

двух понятий. В то время как объект обозначает конкретную сущность, определенную во времени и в пространстве, класс определяет лишь абстракцию существенного в объекте. Таким образом, можно говорить о классе "Млекопитающие", который включает характеристики, общие для всех млекопитающих. Для указания на конкретного представителя млекопитающих необходимо сказать "это - млекопитающее" или "то - млекопитающее".

В общепонятных терминах можно дать следующее определение класса: "группа, множество или вид с общими свойствами или общим свойством, разновидностями, отличиями по качеству, возможностями или условиями" [17]. В контексте объектно-ориентированного анализа дадим следующее определение класса:

Класс - это некое множество объектов, имеющих общую структуру и общее поведение.

Любой конкретный объект является просто экземпляром класса. Что же не является классом? Объект не является классом, хотя в дальнейшем мы увидим, что класс может быть объектом. Объекты, не связанные общностью структуры и поведения, нельзя объединить в класс, так как по определению они не связаны между собой ничем, кроме того, что все они объекты.

Важно отметить, что классы, как их понимают в большинстве существующих языков программирования, необходимы, но не достаточны для декомпозиции сложных систем. Некоторые абстракции так сложны, что не могут быть выражены в терминах простого описания класса. Например, на достаточно высоком уровне абстракции графический интерфейс пользователя,

база данных или система учета как целое, это явные объекты, но не классы.[10] Лучше считать их некими совокупно-стями (кластерами) сотрудничающих классов. Страуструп называет такие кластеры компонентами [18]. Мы же, по причинам, которые будут объяснены в главе 5, называем такие кластеры категориями классов.

Интерфейс и реализация

Мейер [19] и Снайдерс [20] высказали идею контрактного программирования:

большие задачи надо разделить на много маленьких и перепоручить их мелким субподрядчикам. Нигде эта идея не проявляет себя так ярко, как в проектировании классов.

По своей природе, класс - это генеральный контракт между абстракцией и всеми ее клиентами. Выразителем обязательств класса служит его интерфейс, причем в языках с сильной типизацией потенциальные нарушения контракта можно обнаружить уже на стадии компиляции.

Идея контрактного программирования приводит нас к разграничению внешнего облика, то есть интерфейса, и внутреннего устройства класса, реализации. Главное в интерфейсе - объявление операций, поддерживаемых экземплярами класса. К нему можно добавить объявления других классов, переменных, констант и исключительных ситуаций, уточняющих абстракцию, которую класс должен выражать. Напротив, реализация класса никому, кроме него самого, не интересна. По большей части реализация состоит в определении операций, объявленных в интерфейсе класса.

Мы можем разделить интерфейс класса на три части:

• открытую (public) - видимую всем клиентам;

• защищенную (protected) - видимую самому классу, его подклассам и друзьям (friends);

• закрытую (private) - видимую только самому классу и его друзьям.

Разные языки программирования предусматривают различные комбинации этих частей. Разработчик может задать права доступа к той или иной части класса, определив тем самым зону видимости клиента.

В частности, в C++ все три перечисленных уровня доступа определяются явно. В дополнение к этому есть еще и механизм друзей, с помощью которого посторонним классам можно предоставить привилегию видеть закрытую и защищенную области класса. Тем самым нарушается инкапсуляция, поэтому, как и в жизни, друзей надо выбирать осторожно. В Ada объявления могут быть сделаны закрытыми или открытыми. В Smalltalk все переменные - закрыты, а методы - открыты. В Object Pascal все поля и операции открыты, то есть никакой инкапсуляции нет. В CLOS обобщенные функции открыты, а слоты могут быть закрытыми, хотя узнать их значения все равно можно.

Состояние объекта задается в его классе через определения констант или переменных, помещаемые в его защищенной или закрытой части. Тем самым они инкапсулированы, и их изменения не влияют на клиентов.

Внимательный читатель может спросить, почему же представление объекта определяется в интерфейсной части класса, а не в его реализации. Причины чисто практические: в противном случае понадобились бы объектно-ориентированные процессоры или очень хитроумные компиляторы. Когда компилятор обрабатывает объявление объекта, например, такое:

---

[10] Можно попытаться выразить такие абстракции одним классом, но повторной используемости и возможности наследования не получится. Иметь громоздкий интерфейс - плохая практика, так как большинство клиентов использует только малую его часть. Более того, изменение одной части этого гигантского интерфейса требует обновления каждого из клиентов, независимо от того, затронуло ли его это изменение по сути. Вложенность классов не устраняет этих проблем, а только откладывает их.

```
DisplayItem item1;
```

он должен знать, сколько отвести под него памяти. Если бы эта информация содержалась в реализации класса, нам пришлось бы написать ее полностью, прежде, чем мы смогли бы задействовать его клиентов. То есть, весь смысл отделения интерфейса от реализации был бы потерян.

Константы и переменные, составляющие представление класса, известны под разными именами. В Smalltalk их называют переменные экземпляра, в Object Pascal - поля, в C++ - члены класса, а в CLOS - слоты. Мы часто будем использовать эти термины как синонимы.

Жизненный цикл класса

В поведении простых классов можно разобраться, изучая операции их открытой части. Однако поведение более интересных классов (такое как перемещение объекта класса **DisplayItem** или составление расписания для экземпляра класса **TemperatureController**) включает взаимодействие разных операций, входящих в класс. Как уже говорилось выше, объекты таких классов действуют как маленькие машины, части которых взаимодействуют друг с другом, и так как все такие объекты имеют одно и то же поведение, можно использовать класс для описания их общей семантики, упорядоченной по времени и событиям. Как будет показано в главе 5, мы можем описывать динамику поведения объектов, используя модель конечного автомата.

3.4. Отношения между классами

Типы отношений

Рассмотрим сходства и различия между следующими классами: цветы, маргаритки, красные розы, желтые розы, лепестки и божьи коровки. Мы можем заметить следующее:

• Маргаритка - цветок.

• Роза - (другой) цветок.

• Красная и желтая розы - розы.

• Лепесток является частью обоих видов цветов.

• Божьи коровки питаются вредителями, поражающими некоторые цветы.

Из этого простого примера следует, что классы, как и объекты, не существуют изолированно. В каждой проблемной области ключевые абстракции взаимодействуют многими интересными способами, что мы и должны отразить в проекте [21].

Отношения между классами могут означать одно из двух. Во-первых, у них может быть что-то общее. Например, и маргаритки, и розы - это разновидности цветов: и те, и другие имеют ярко окрашенные лепестки, испускают аромат и так далее. Во-вторых, может быть какая-то семантическая связь. Например, красные розы больше похожи на желтые розы, чем на маргаритки. Но между розами и маргаритками больше общего, чем между цветами и лепестками. Также существует сим-биотическая связь между цветами и божьими коровками: божьи коровки защищают цветы от вредителей, которые, в свою очередь, служат пищей божьим коровкам.

Известны три основных типа отношений между классами [22]. Во-первых, это отношение "обобщение/специализация" (общее и частное), известное как "is-a". Розы суть цветы, что значит: розы являются специализированным частным случаем, подклассом более общего класса "цветы". Во вторых, это отношение "целое/ часть", известное как "part of". Так, лепестки являются частью цветов. В-третьих, это семантические, смысловые отношения, ассоциации. Например, божьи коровки ассоциируются с цветами -

хотя, казалось бы, что у них общего. Или вот: розы и свечи - и то, и другое можно использовать для украшения стола.

Языки программирования выработали несколько общих подходов к выражению отношений этих трех типов. В частности, большинство объектно-ориентированных языков непосредственно поддерживают разные комбинации следующих видов отношений:

- ассоциация
- наследование
- агрегация
- использование
- инстанцирование
- метакласс.

Альтернативой наследованию является делегирование, при этом объекты рассматриваются как прототипы, которые делегируют свое поведение родственным им объектам. Таким образом, классы становятся не нужны [23].

Из шести перечисленных видов отношений наиболее общим и неопределенным является ассоциация. Как мы увидим в главе 6, обычно аналитик констатирует наличие ассоциации и, постепенно уточняя проект, превращает ее в какую-то более специализированную связь.

Наследование, вероятно, следует считать самым интересным семантически. Оно выражает отношение общего и частного. Однако, по нашему опыту, одного наследования недостаточно, чтобы выразить все многообразие явлений и отношений жизни. Полезна также агрегация, отражающая отношения целого и части между экземплярами классов. Нелишне добавить отношение использования, означающее наличие связи между экземплярами классов. Имея дело с языками Ada, Eiffel и C++, нам не обойтись без инстанцирования, которое, подобно наследованию, является специфической разновидностью обобщения. "Метаклассовые" отношения - это нечто совершенно иное, в явном виде встречающееся только в языках Smalltalk

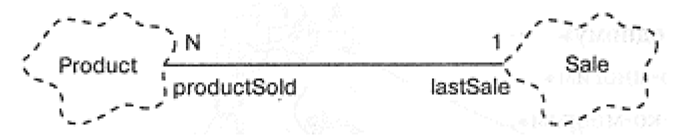


Рис. 3-4. Ассоциация

и CLOS. Метакласс - это класс классов, что позволяет нам трактовать классы как объекты.

Ассоциация

Пример. Желая автоматизировать розничную торговую точку, мы обнаруживаем две абстракции - товары и продажи. На рис. 3-4 показана ассоциация, которую мы при этом усматриваем. Класс `Product` - это то, что мы продали в некоторой сделке, а класс `Sale` - сама сделка, в которой продано несколько товаров. Надо полагать, ассоциация работает в обе стороны: задавшись товаром, можно выйти на сделку, в которой он был продан, а пойдя от сделки, найти, что было продано.

В C++ это можно выразить с помощью того, что Румбах называет погребенными указателями [24]. Вот две выдержки из объявления соответствующих классов:

```
class Product;

class Sale;

class Product {
public:
```

```
…
protected:
	Sale* lastSale;
};

class Sale {
public:
…
protected:
	Product** productSold;
};
```

Это ассоциация вида "один-ко-многим": каждый экземпляр товара относится только к одной последней продаже, в то время как каждый экземпляр **Sale** может указывать на совокупность проданных товаров.

Семантические зависимости. Как показывает этот пример, ассоциация - смысловая связь. По умолчанию, она не имеет направления (если не оговорено противное, ассоциация, как в данном примере, подразумевает двухстороннюю связь) и не объясняет, как классы общаются друг с другом (мы можем только отметить семантическую зависимость, указав, какие роли классы играют друг для друга). Однако именно это нам требуется на ранней стадии анализа. Итак, мы фиксируем участников, их роли и (как будет сказано далее) мощность отношения.

Мощность. В предыдущем примере мы имели ассоциацию "один ко многим". Тем самым мы обозначили ее мощность (то есть, грубо говоря, количество участников). На практике важно различать три случая мощности ассоциации:

- "один-к-одному"
- "один-ко-многим"
- "многие-ко-многим".

Отношение "один-к-одному" обозначает очень узкую ассоциацию. Например, в розничной системе продаж примером могла бы быть связь между классом **Sale** и классом **CreditCardTransaction**: каждая продажа соответствует ровно одному снятию денег с данной кредитной карточки. Отношение "многие-ко-многим" тоже нередки. Например, каждый объект класса **Customer** (покупатель) может инициировать транзакцию с несколькими объектами класса **Saleperson** (торговый агент), и каждый торговый агент может взаимодействовать с несколькими покупателями. Как мы увидим в главе 5, все три вида мощности имеют разного рода вариации.

Наследование

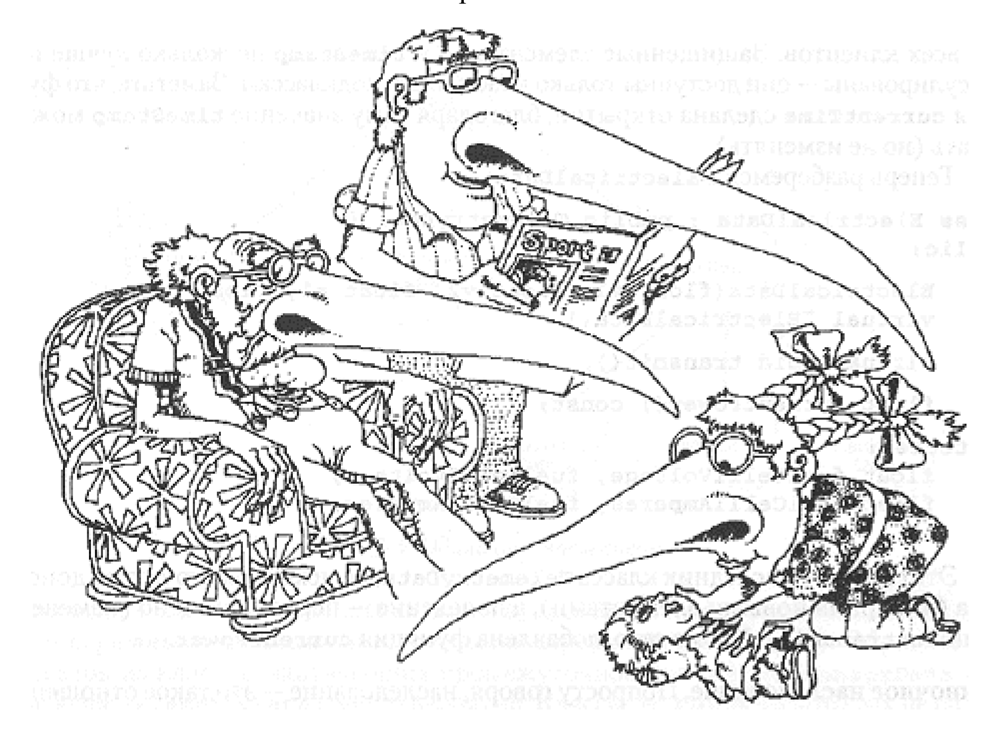
Примеры. Находящиеся в полете космические зонды посылают на наземные станции информацию о состоянии своих основных систем (например, источников энергоснабжения и двигателей) и измерения датчиков (таких как датчики радиации, масс-спектрометры, телекамеры, фиксаторы столкновений с микрометеоритами и т. д.). Вся совокупность передаваемой информации называется телеметрическими данными. Как правило, они передаются в виде потока данных, состоящего из заголовка (включающего временные метки и ключи для идентификации последующих данных) и нескольких пакетов данных от подсистем и датчиков. Все это выглядит как простой набор разнотипных данных, поэтому для описания каждого типа данных телеметрии сами собой напрашиваются структуры:

```
class Time...

struct ElectricalData {
	Time timeStamp;
	int id;
```

```
        float fuelCell1Voltage, fuelCell2Voltage;
        float fuelCell1Amperes, fuelCell2Amperes;
        float currentPower;
};
```

Однако такое описание имеет ряд недостатков. Во-первых, структура класса **ElectricalData** не защищена, то есть клиент может вызвать изменение такой важной информации, как **timeStamp** или **currentPower** (мощность, развиваемая обеими электробатареями, которую можно вычислить из тока и напряжения). Во-вторых, эта структура является полностью открытой, то есть ее модификации (добавление новых элементов в структуру или изменение типа существующих элементов) влияют на клиентов. Как минимум, приходится заново компилировать все описания, связанные каким-либо образом с этой структурой. Еще важнее, что внесение в структуру изменений может нарушить логику отношений с клиентами, а следовательно, логику всей программы. Кроме того, приведенное описание структуры очень трудно для восприятия. По отношению к такой структуре можно выполнить множество различных действий (пересылка данных, вычисление контрольной суммы для определения ошибок и т. д.), но все они не будут связаны с приведенной структурой логически. Наконец, предположим, что анализ требова-



Дочерний класс может унаследовать структуру и поведение родительских классов

ний к системе обусловил наличие нескольких сотен разновидностей телеметрических данных, включающих показанную выше структуру и другие электрические параметры в разных контрольных точках системы. Очевидно, что описание такого количества дополнительных структур будет избыточным как из-за повторяемости структур, так и из-за наличия общих функций обработки.

Лучше было бы создать для каждого вида телеметрических данных отдельный класс, что позволит защитить данные в каждом классе и увязать их с выполняемыми операциями. Но этот подход не решает проблему избыточности.

Значительно лучше построить иерархию классов, в которой от общих классов с помощью наследования образуются более специализированные; например, следующим образом:

```
class TelemetryData {
public:
      TelemetryData() ;
      virtual ~TelemetryData();
      virtual void transmit() ;
      Time currentTime() const;
protected:
      int id;
      Time timeStamp;
};
```

В этом примере введен класс, имеющий конструктор, деструктор (который иаследники могут переопределить) и функции **transmit** и **currentTime**, видимые для всех клиентов. Защищенные элементы **id** и **timeStamp** несколько лучше инкапсулированы - они доступны только классу и его подклассам. Заметьте, что функция currentTlrne сделана открытой, благодаря чему значение **timeStamp** можно читать (но не изменять).

Теперь разберемся с **ElectricalData**:

```
class ElectricalData : public TelemetryData {
public:
      ElectricalData(float vl, float v2, float al, float a2);
      virtual ~ElectricalData();
      virtual void. transmit();
      float currentPower () const;
protected:
      float fuelCelllVoltage, fuelCell2Voltage;
      float fuelCelllAmperes, fuelCell2Amperes;
};
```

Этот класс - наследник класса **TelemetryData**, но исходная структура дополнена (четырьмя новыми элементами), а поведение - переопределено (изменена функция **transmit**). Кроме того, добавлена функция **currentPower**.

Одиночное наследование. Попросту говоря, наследование - это такое отношение между классами, когда один класс повторяет структуру и поведение другого класса (одиночное наследование) или других (множественное наследование) классов. Класс, структура и поведение которого наследуются, называется суперклассом. Так, **TelemetryData**. является суперклассом для **ElectricalData**. Производный от суперкласса класс называется подклассом. Это означает, что наследование устанавливает между классами иерархию общего и частного. В этом смысле **ElectricalData** является более специализированным классом более общего **TelemetryData**. Мы уже видели, что в подклассе структура и поведение исходного суперкласса дополняются и переопределяются. Наличие механизма наследования отличает объектно-ориентированные языки от объектных.

Подкласс обычно расширяет или ограничивает существующую структуру и поведение своего суперкласса. Например, подкласс **GuardedQueue** может добавлять к поведению суперкласса **Queue** операции, которые защищают состояние очереди от одновременного изменения несколькими независимыми потоками. Обратный пример: подкласс **UnselectableDisplayltem** может ограничить поведение своего суперкласса **Displayitem**, запретив выделение объекта на экране. Часто подклассы делают и то, и другое.

Отношения одиночного наследования от суперкласса **TelemetryData** показаны на рис. 3-5. Стрелки обозначают отношения общего к частному. В частности, **Cameradata** - это разновидность класса **SensorData**, который в свою очередь является разновидностью класса **TelemetryData**. Такой же тип иерархии характерен для семантических сетей, которые часто используются специалистами по распознаванию образов и искусственному интеллекту для

организации баз знаний [25]. В главе 4 мы покажем, что правильная организация иерархии абстракций - это вопрос логической классификации.

Можно ожидать, что для некоторых классов на рис. 3-5 будут созданы экземпляры, а для других - нет. Наиболее вероятно образование объектов самых специа-

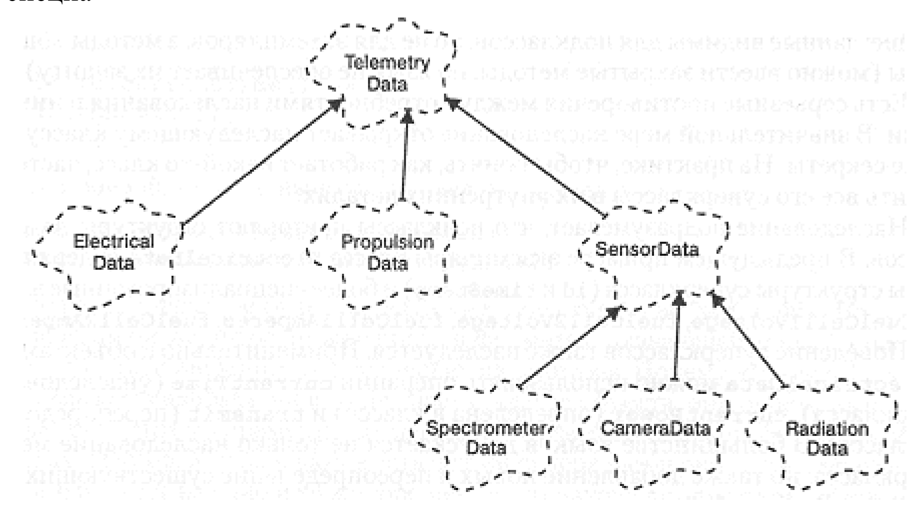


Рис. 3-5. Одиночное наследование

лизированных классов **ElectricalData** и **SpectrometerData** (такие классы называют конкретными классами, или листьями иерархического дерева). Образование объектов из классов, занимающих промежуточное положение (**SensorData** или тем более **TelemetryData**), менее вероятно. Классы, экземпляры которых не создаются, называются абстрактными. Ожидается, что подклассы абстрактных классов доопределят их до жизнеспособной абстракции, наполняя класс содержанием. В языке Smalltalk разработчик может заставить подкласс переопределить метод, помещая в реализацию метода суперкласса вызов метода **SubclassResponsibility**. Если метод не переопределен, то при попытке выполнить его генерируется ошибка. Аналогично, в C++ существует возможность объявлять функции чисто виртуальными. Если они не переопределены, экземпляр такого класса невозможно создать.

Самый общий класс в иерархии классов называется базовым. В большинстве приложений базовых классов бывает несколько, и они отражают наиболее общие абстракции предметной области. На самом деле, особенно в C++, хорошо сделанная структура классов - это скорее лес из деревьев наследования, чем одна многоэтажная структура наследования с одним корнем. Но в некоторых языках программирования определен базовый класс самого верхнего уровня, который является единым суперклассом для всех остальных классов. В языке Smalltalk эту роль играет класс object.

У класса обычно бывает два вида клиентов [26]:

- экземпляры
- подклассы.

Часто полезно иметь для них разные интерфейсы [27]. В частности, мы хотим показать только внешне видимое поведение для клиентов-экземпляров, но нам нужно открыть служебные функции и представления клиентам-подклассам. Этим объясняется наличие открытой, защищенной и закрытой частей описания класса в языке C++: разработчик может четко разделить, какие элементы класса доступны Для экземпляров, а какие для подклассов. В языке Smalltalk степень такого разделения меньше: данные видимы для подклассов, но не для экземпляров, а методы общедоступны (можно ввести закрытые методы, но язык не обеспечивает их защиту).

Есть серьезные противоречия между потребностями наследования и инкапсуляции. В значительной мере наследование открывает наследующему

классу некоторые секреты. На практике, чтобы понять, как работает какой-то класс, часто надо изучить все его суперклассы в их внутренних деталях.

Наследование подразумевает, что подклассы повторяют структуры их суперклассов. В предыдущем примере экземпляры класса **ElectricalData** содержат элементы структуры суперкласса (**id** и **timeStamp**) и более специализированные элементы (**fuelCell1Voltage, fuelCell2Voltage, fuelCell1Amperes, fuelCell2Amperes**).[11]

Поведение суперклассов также наследуется. Применительно к объектам класса **ElectricalData** можно использовать операции **currentTime** (унаследована от суперкласса), **currentPower** (определена в классе) и **transmit** (переопределена в подклассе). В большинстве языков допускается не только наследование методов суперкласса, но также добавление новых и переопределение существующих методов. В Smalltalk любой метод суперкласса можно переопределить в подклассе.

В C++ степень контроля за этим несколько выше. Функция, объявленная виртуальной (функция **transmit** в предыдущем примере), может быть в подклассе переопределена, а остальные (**currentTime**) - нет.

Одиночный полиморфизм. Пусть функция **transmit** класса **TelemetryData** реализована следующим образом:

```
void TelemetryData::transmit()
{
      // передать id
      // передать timeStamp
};
```

В классе **ElectricalData** та же функция переопределена:

```
void ElectricalData::transmit()
{
      TelemetryData::transmit();
      // передать напряжение
      // передать силу тока
};
```

Эта функция сначала вызывает одноименную функцию суперкласса с помощью ее явно квалифицированного имени **TelemetryData**:: **transmit** (). Та передаст заголовок пакета (**id** и **timeStamp**), после чего в подклассе передаются его собственные данные.

Определим теперь экземпляры двух описанных выше классов:

```
TelemetryData telemetry;
ElectricalData electrical(5.O, -5.0, 3.0, 7.0);
```

Теперь определим свободную процедуру:

```
void transmitFreshData (TelemetryData& d, const Time& t)
{
      if (d.currentTime() >= t)
      d.transmit() ;
);
```

Что произойдет, если выполнить следующие два оператора?

```
transmitFreshData(telemetry, Time(60));
transmitFreshData(electrical, Time(120));
```

В первом операторе будет передан уже известный нам заголовок. Во втором будет передан он же, плюс четыре числа в формате с плавающей точкой, содержащие результаты измерений электрических параметров. Почему это так? Ведь функция **transmitFreshData** ничего не знает о классе объекта, она просто выполняет d,**transmit** ()! Это был пример полиморфизма. Переменная d может обозначать объекты разных классов. У

---

[11] Некоторые языки объектно-ориентированного программирования, главным образом экспериментальные, позволяют подклассу сокращать структуру его суперкласса.

этих классов есть общий суперкласс и они, хотя и по разному, могут реагировать на одно и то же сообщение, одинаково понимая его смысл.

Карделли и Вегнер заметили, что "традиционные типизированные языки типа Pascal основаны на той идее, что функции и процедуры, а следовательно, и операнды должны иметь определенный тип. Это свойство называется мономорфизмом, то есть каждая переменная и каждое значение относятся к одному определенному типу. В противоположность мономорфизму полиморфизм допускает отнесение значений и переменных к нескольким типам" [28]. Впервые идею полиморфизма ad hoc описал Страчи [29], имея в виду возможность переопределять смысл симво-лов, таких, как "+", сообразно потребности. В современном программировании мы называем это перегрузкой. Например, в C++ можно определить несколько функций с одним и тем же именем, и они будут автоматически различаться по количеству и типам своих аргументов. Совокупность этих признаков называется сигнатурой функции; в языке Ada к этому списку добавляется тип возвращаемого значения. Страчи говорил также о параметрическом полиморфизме, который мы сейчас называем просто полиморфизмом.

При отсутствии полиморфизма код программы вынуждено содержит множество операторов выбора case или switch. Например, на языке Pascal невозможно образовать иерархию классов телеметрических данных; вместо этого придется определить одну большую запись с вариантами, включающую все разновидности данных. Для выбора варианта нужно проверить метку, определяющую тип записи. На языке Pascal процедура **TransmitFreshData** может быть написана следующим образом:

```
const
      Electrical = 1;
      Propulsion =2;
      Spectrometer = 3;

Procedure Transmit_Presh_Data(TheData : Data; The_Tiime :
Time);
      begin
            if (The_Data.Current_Time >= The_Time)
            then    case TheData.Kind of
                  Electrical:
                  Transmit_Electrical_Data(The_Data);
                  Propulsion:
            Transmit_Propulsion_Data(The_Data);
                  end
end;
```

Чтобы ввести новый тип телеметрических данных, нужно модифицировать эту вариантную запись, добавив новый тип в каждый оператор case. В такой ситуации увеличивается вероятность ошибок, и проект становится нестабильным.

Наследование позволяет различать разновидности абстракций, и монолитные типы становятся не нужны. Каплан и Джонсон отметили, что "полиморфизм наиболее целесообразен в тех случаях, когда несколько классов имеют одинаковые протоколы" [30]. Полиморфизм позволяет обойтись без операторов выбора, поскольку объекты сами знают свой тип.

Наследование без полиморфизма возможно, но не очень полезно. Это видно на примере Ada, где можно объявлять производные типы, но из-за мономорфизма языка операции жестко задаются на стадии компиляции.

Полиморфизм тесно связан с механизмом позднего связывания. При полиморфизме связь метода и имени определяется только в процессе выполнения программ. В C++ программист имеет возможность выбирать между ранним и поздним связыванием имени с операцией. Если функция

виртуальная, связывание будет поздним и, следовательно, функция полиморфна. Если нет, то связывание происходит при компиляции и ничего изменить потом нельзя. Этому вопросу посвящена следующая врезка.

Наследование и типизация. Рассмотрим еще раз переопределение функции **transmit**:

```
void ElectricalData::transmit()
{
      TelemetryData:: transmit ();
      // передать напряжение
      // передать силу тока
};
```

В большинстве объектно-ориентированных языков программирования при реализации метода подкласса разрешается вызывать напрямую метод какого-либо суперкласса. Как видно из примера, это допускается и в том случае, если метод подкласса имеет такое же имя и фактически переопределяет метод суперкласса. В Smalltalk метод вышестоящего класса вызывают с помощью ключевого слова super, при этом вызывающий может указывать на самого себя с помощью ключевого слова self. В C++ метод любого достижимого вышестоящего класса можно вызывать, добавляя имя класса в качестве префикса, формируя квалифицированное имя метода (как **TelemetryData**:: **transmit** () в нашем примере). Вызывающий объект может ссылаться на себя с помощью предопределенного указателя this.

На практике метод суперкласса вызывается до или после дополнительных действий. Метод подкласса уточняет или дополняет поведение суперкласса.[12]

Все подклассы на рис. 3-5 являются также подтипами вышестоящего класса. В частности, **ElectricalData** является подтипом **TelemetryData**. Система типов, развивающаяся параллельно наследованию, обычна для объектно-ориентирован-

Поиск метода

Рассмотрим иерархию (рис. 3-6), в которой имеется базовый класс и три подкласса с именами circle, Triangle и Rectangle. Для класса Rectangle определен в свою оче-редь подкласс SolidRectangle. Предположим, что в классе DisplayItem определена переменная экземпляра theCenter (задающая координаты центра изображения), а также следующие операции:

- draw - нарисовать изображение
- move - передвинуть изображение
- **location** - вернуть координаты изображения.

Операция **location** является общей для всех подклассов и не требует обязательного переопределения. Однако, поскольку только подклассы могут знать, как их изображать и передвигать, операции draw и move должны быть переопределены.

---

[12] В CLOS эти различные роли метода выражаются явно с помощью дополнительных квалификаторов : before,: after или : around. Метод без дополнительного квалификатора считается первичным и выполняет основную работу, обеспечивающую требуемое поведение. Before-метод вызывается до первичного, after-метод - после первичного, around-ме-тод действует как оболочка вокруг первичного метода, которая вызывается изнутри этого метода функцией call-next-method.

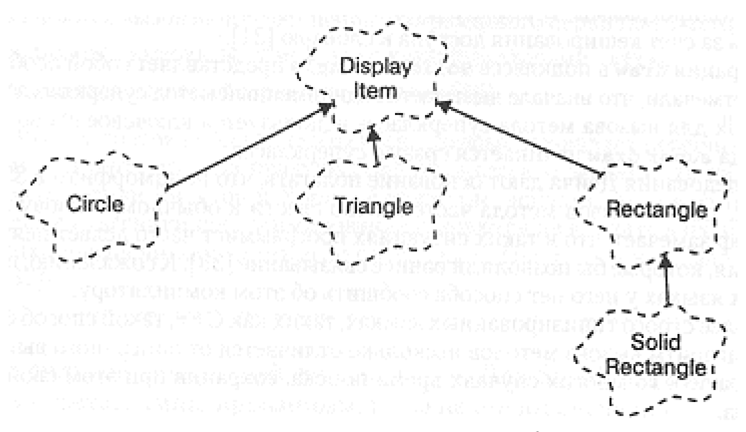


Рис. 3-6. Диаграмма класса Display Item

Класс Circle имеет переменную theRadius и соответственно операции для установки (set) и чтения значения этой переменной. Для этого класса операция draw формирует изображение окружности заданного радиуса с центром в точке theCenter. В классе Rectangle есть переменные theHeight и theWidth и соответствующие операции установки и чтения их значений. Операция draw в данном случае формирует изображение прямоугольника заданной высоты и ширины с центром в заданной точке theCenter. Подкласс SolidRectangle наследует все особенности класса Rectangle, но операция draw в этом подклассе переопределена. Сначала вызывается draw вышестоящего класса, а затем полученный контур заполняется цветом.

Теперь рассмотрим следующий фрагмент программы:

```
DisplayItem* items[10] ;
for (unsigned index = 0; index < 10; index++) items[index]->draw() ;
```

Вызов draw требует полиморфного поведения. У нас есть разнородный массив объектов, содержащий указатели на любые разновидности DisplayItem. Пусть некоторый клиент хочет, чтобы они все изобразили себя на экране. Наш подход - перебрать элементы массива и послать каждому указываемому объекту сообщение draw. Компилятор не может определить, какую функцию и откуда надо при этом вызвать, так как невозможно предсказать, на что будут указывать элементы массива во время выполнения программы. Посмотрим, как эта задача будет решаться в разных объектно-ориентированных языках.

Поскольку в Smalltalk нет типов, методы вызываются строго динамически. Когда клиент посылает сообщение draw очередному получателю, происходит следующее:

• получатель ищет селектор сообщения в словаре методов своего класса;

• если метод найден, то запускается его код;

• если нет, поиск производится в словаре методов суперкласса.

Таким образом, поиск распространяется вверх по иерархии и заканчивается на классе object, который является "предком" всех классов. Если метод не найден и там, посылается сообщение doesNotUnderstand, то есть, генерируется ошибка.

Главным действующим лицом в этом алгоритме является словарь методов. Он формируется при создании класса и, являясь частью его реализации, скрыт от клиентов. Вызов метода в Smalltalk требует примерно в 1.5 раза больше времени, чем вызов простой подпрограммы. В коммерческих

версиях Smalltalk вызов методов ускорен на 20-30% за счет кеширования доступа к словарю [31].

Операция draw в подклассе solidRectangle представляет собой особый случай. Мы уже отмечали, что вначале вызывается одноименный метод суперкласса Rectangle. В Smalltalk для вызова метода суперкласса используется ключевое слово super. Поиск метода super draw начинается сразу с суперкласса.

Исследования Дейча дают основание полагать, что полиморфизм в 85% случаев не нужен, так что вызов метода часто можно свести к обычному вызову процедуры [32]. Дафф замечает, что в таких ситуациях программист часто делает неявные предположения, которые бы позволили раннее связывание [33]. К сожалению, в нетипизированных языках у него нет способа сообщить об этом компилятору.

В более строго типизированных языках, таких как C++, такой способ есть. В этих языках алгоритм вызова методов несколько отличается от описанного выше и позволяет сократить во многих случаях время поиска, сохранив при этом свойства полиморфизма.

В C++ операции для позднего связывания объявляются виртуальными (virtual), а все остальные обрабатываются компилятором как обычные вызовы подпрограмм. В нашем примере draw - виртуальная функция, а **location** - обычная. Есть еще одно средство, используя которое можно выиграть в скорости. Невиртуальные методы могут быть объявлены подставляемыми (inline), при этом соответствующая подпрограмма не вызывается, а явно включается в код на манер макроподстановки.

Для управления виртуальными функциями в C++ используется концепция vtable (виртуальных таблиц), которые формируются для каждого объекта при его создании (то есть когда класс объекта уже известен). Такая таблица содержит список указателей на виртуальные функции. Например, при создании объекта класса Rectangle виртуальная таблица будет содержать запись для виртуальной функции draw, содержащую указатель на ближайшую в иерархии реализацию функции draw. Если в классе DisplayItem есть виртуальная функция rotate, которая в классе Rectangle не переопределена, то соответствующий указатель для rotate останется связан с классом DisplayItem. Во время исполнения программы происходит косвенное обращение через соответствующий указатель и сразу выполняется нужный код без всякого поиска

[34].

Операция draw в классе SolidRectangle представляет собой особый случай и в языке C++. Чтобы вызвать метод draw из суперкласса, применяется специальный префикс, указывающий на место определения функции. Это выглядит следующим образом:

Rectangle::Draw();

Исследование Страуструпа показало, что вызов виртуальной функции по эффективности мало уступает вызову обычной функции [35]. Для одиночного наследования вызов виртуальной функции требует дополнительно выполнения трех-четырех операций доступа к памяти по отношению к обычному вызову; при множественном наследовании число таких дополнительных операций составляет пять или шесть.

Существенно сложнее выполняется поиск нужных функций в языке CLOS, здесь используются дополнительные квалификаторы: : before, : after, : around. Наличие множественного полиморфизма еще более усложняет проблему. При вызове метода в языке CLOS, как правило, реализуется следующий алгоритм:

• Определяется тип аргументов.

• Вычисляется множество допустимых методов.

• Методы сортируются в направлении от наиболее специализированных к более общим.

• Выполняются вызовы всех методов с квалификатором : before.

• Выполняется вызов наиболее специализированного первичного метода.

• Выполняются вызовы всех методов с квалификаторами : after.

• Возвращается значение первичного метода [36].

В CLOS есть протокол для метаобъектов, который позволяет переопределять в том числе и этот алгоритм. На практике, однако, мало кто заходит так далеко. Как справедливо отметили Винстон и Хорн: "Алгоритмы, используемые в языке CLOS, сложны, и даже кудесники программирования стараются не вникать в их детали, так же как физики предпочитают иметь дело с механикой Ньютона, а не с квантовой механикой" [37].

ных языков с сильной типизацией, включая C++. Для Smalltalk, который едва ли вообще можно считать типизированным, типы не имеют значения.

Параллелей между типизацией и наследованием следует ожидать там, где иерархия общего и частного предназначена для выражения смысловых связей между абстракциями. Рассмотрим снова объявления в C++:

```
TelenetryData telemetry;
ElectrycalData electrical (5.0, -5.0, 3.0, 7.0);
```

Следующий оператор присваивания правомочен:

```
telemetry = electrical; //electrical - это подтип telemetry
```

Хотя он формально правилен, он опасен: любые дополнения в состоянии подкласса по сравнению с состоянием суперкласса просто срезаются. Таким образом, Дополнительные четыре параметра, определенные в подклассе electrical, будут потеряны при копировании, поскольку их просто некуда записать в объекте telemetry класса **TelemetryData**.

Следующий оператор неправилен:

```
electrical= telemetry; //неправильно: telemetry-это не подтип electrical
```

Можно сделать заключение, что присвоение объекту у значения объекта х до-пустимо, если тип объекта х совпадает с типом объекта у или является его подтипом.

В большинстве строго типизированных языков программирования допускается преобразование значений из одного типа в другой, но только в тех случаях, когда между двумя типами существует отношение класс/подкласс. В языке C++ есть оператор явного преобразования, называемый приведением типов. Как правило, такие преобразования используются по отношению к объекту специализированного класса, чтобы присвоить его значение объекту более общего класса. Такое приведение типов считается безопасным, поскольку во время компиляции осуществляется семантический контроль. Иногда необходимы операции приведения объектов более общего класса к специализированным классам. Эти операции не являются надежными с точки зрения строгой типизации, так как во время выполнения программы может возникнуть несоответствие (несовместимость) приводимого объекта с новым типом.[13] Однако такие преобразования достаточно часто используются в тех случаях, когда программист хорошо представляет себе все типы объектов. Например, если нет параметризованных типов, часто создаются классы set или bag, представляющие собой наборы произвольных объектов. Их определяют для некоторого базового класса (это гораздо безопаснее, чем использовать идиому void*, как мы делали, определяя класс Queue). Итерационные операции, определенные для такого

---

[13] Новейшие усовершенствования C++, направленные на динамическое определение типа, смягчили эту проблему.

класса, умеют возвращать только объекты этого базового класса. Внутри конкретного приложения разработчик может использовать этот класс, создавая объекты только какого-то специализированного подкласса, и, зная, что именно он собирается помещать в этот класс, может написать соответствующий преобразователь. Но вся эта стройная конструкция рухнет во время выполнения, если в наборе встретится какой-либо объект неожиданного типа.

Большинство сильно типизированных языков позволяют приложениям оптимизировать технику вызова методов, зачастую сводя пересылку сообщения к простому вызову процедуры. Если, как в C++, иерархия типов совпадает с иерархией классов, такая оптимизация очевидна. Но у нее есть недостатки. Изменение структуры или поведения какого-нибудь суперкласса может поставить вне закона его подклассы. Вот что об этом пишет Микаллеф: "Если правила образования типов основаны на наследовании и мы переделываем какой-нибудь класс так, что меняется его положение в иерархии наследования, клиенты этого класса могут оказаться вне закона с точки зрения типов, несмотря на то, что внешний интерфейс класса остается прежним" [38].

Тем самым мы подходим к фундаментальным вопросам наследования. Как было сказано выше, наследование используется в связи с тем, что у объектов есть что-то общее или между ними есть смысловая ассоциация. Выражая ту же мысль иными словами, Снайдерс пишет: "наследование можно рассматривать, как способ управления повторным использованием программ, то есть, как простое решение разработчика о заимствовании полезного кода. В этом случае механика наследования должна быть гибкой и легко перестраиваемой. Другая точка зрения:

наследование отражает принципиальную родственность абстракций, которую невозможно отменить" [39]. В Smalltalk и CLOS эти два аспекта неразделимы. C++ более гибок. В частности, при определении класса его суперкласс можно объявить public (как **ElectricalData** в нашем примере). В этом случае подкласс считается также и подтипом, то есть обязуется выполнять все обязательства суперкласса, в частности обеспечивая совместимое с суперклассом подмножество интерфейса и обладая неразличимым с точки зрения клиентов суперкласса поведением. Но если при определении класса объявить его суперкласс как private, это будет означать, что, наследуя структуру и поведение суперкласса, подкласс уже не будет его под-типом.[14] Это означает, что открытые и защищенные члены суперкласса станут закрытыми членами подкласса, и следовательно они будут недоступны подклассам более низкого уровня. Кроме того, тот факт, что подкласс не будет подтипом, означает, что класс и суперкласс обладают несовместимыми (вообще говоря) интерфейсами с точки зрения клиента. Определим новый класс:

```
class InternalElectricalData : private ElectricalData {
public:
      InternalElectricalData(float vl, float v2, float al,
float a2);
      virtual "InternalElectricalData() ;
      ElectricalData::currentPower;
};
```

Здесь суперкласс **ElectricalData** объявлен закрытым. Следовательно, его методы, такие, например, как **transmit**, недоступны клиентам. Поскольку класс **InternalElectricalData** не является подтипом **ElectricalData**, мы уже не сможем присваивать экземпляры подкласса объектам суперкласса, как в случае объявления суперкласса в качестве

---

[14] Мы можем также объявить суперкласс защищенным, что даст ту же семантику, что и в случае закрытого суперкласса, но открытые и защищенные элементы такого суперкласса будут доступны подклассам.

открытого. Отметим, что функция **currentPower** сделана видимой путем ее явной квалификации. Иначе она осталась бы закрытой. Как можно было ожидать, правила С++ запрещают делать унаследованный элемент в подклассе "более открытым", чем в суперклассе. Так, член **timeStamp**, объявленный в классе **TelemetryData** защищенным, не может быть сделан в подклассе открытым путем явного упоминания (как это было сделано для функции **currentpower**).

В языке Ada для достижения аналогичного эффекта вместо подтипов используется механизм производных типов. Определение подтипа означает не появление нового типа, а лишь ограничение существующего. А вот определение производного типа создает самостоятельный новый тип, который имеет структуру, заимствованную у исходного типа.

В следующем разделе мы покажем, что наследование с целью повторного использования и агрегация до некоторой степени противоречат друг другу.

Множественное наследование. Мы рассмотрели вопросы, связанные с одиночным наследованием, то есть, когда подкласс имеет ровно один суперкласс. Однако, как указали Влиссидес и Линтон: "одиночное наследование при всей своей полезности часто заставляет программиста выбирать между двумя равно привлекательными классами. Это ограничивает возможность повторного использования предопределенных классов и заставляет дублировать уже имеющиеся коды. Например, нельзя унаследовать графический элемент, который был бы одновременно окружностью и картинкой; приходится наследовать что-то одно и добавлять необходимое от второго" [40].

Множественное наследование прямо поддерживается в языках C++ и CLOS, а также, до некоторой степени, в Smalltalk. Необходимость множественного наследования в OOP остается предметом горячих споров. По нашему опыту, множественное наследование - как парашют: как правило, он не нужен, но, когда вдруг он понадобится, будет жаль, если его не окажется под рукой.

Представьте себе, что нам надо организовать учет различных видов материального и нематериального имущества - банковских счетов, недвижимости, акций и облигаций. Банковские счета бывают текущие и сберегательные. Акции и облигации можно отнести к ценным бумагам, управление ими совершенно отлично от банковских счетов, но и счета и ценные бумаги - это разновидности имущества.

Однако есть много других полезных классификаций тех же видов имущества. В каком-то контексте может потребоваться отличать то, что можно застраховать (недвижимость и, до некоторой степени, сберегательные вклады). Другой аспект - способность имущества приносить дивиденды; это общее свойство банковских счетов и ценных бумаг.

Очевидно, одиночное наследование в данном случае не отражает реальности, так что придется прибегнуть к множественному[15]. Получившаяся структура классов показана на рис. 3-7. На нем класс **Security** (ценные бумаги) наследует одновременно от классов **InterestBearingltem** (источник дивидендов) и **Asset** (имущество). Сходным образом,

---

[15] В действительности, это - "лакмусовая бумажка" для множественного наследования. Если мы составим структуру классов, в которой конечные классы (листья) могут быть сгруппированы в множества по разным ортогональным признакам (как в нашем примере, где такими признаками были способность приносить дивиденды и возможность страховки) и эти множества перекрываются, то это служит признаком невозможности обойтись одной структурой наследования, в которой бы существовали какие-то промежуточные классы с нужным поведением. Мы можем исправить ситуацию, используя множественное наследование, чтобы соединить два нужных поведения там, где это необходимо.

BankAccount (банковский счет) наследует сразу от трех классов: **InsurableItem** (страхуемое) и уже известным **Asset** и **InterestBearingItem**.

Вот как это выражается на C++. Сначала базовые классы:

```
class Asset ...
class InsurableItem ...
class InterestBearingItem ...
```

Теперь промежуточные классы; каждый наследует от нескольких суперклассов:

```
class BankAccount :     public Asset,
                        public InsurableItem ,
                        public InterestBearingItem ...
class RealEstate :      public Asset,
                        public InsurableItem ...
class Security :        public Asset,
                        public InterestBearingItem ...
```

Наконец,листья:

```
class SavingsAccount : public BankAccount . . .
class CheckingAccount : public BankAccount ...

class Stock : public Security ...
class Bond : public Security ...
```

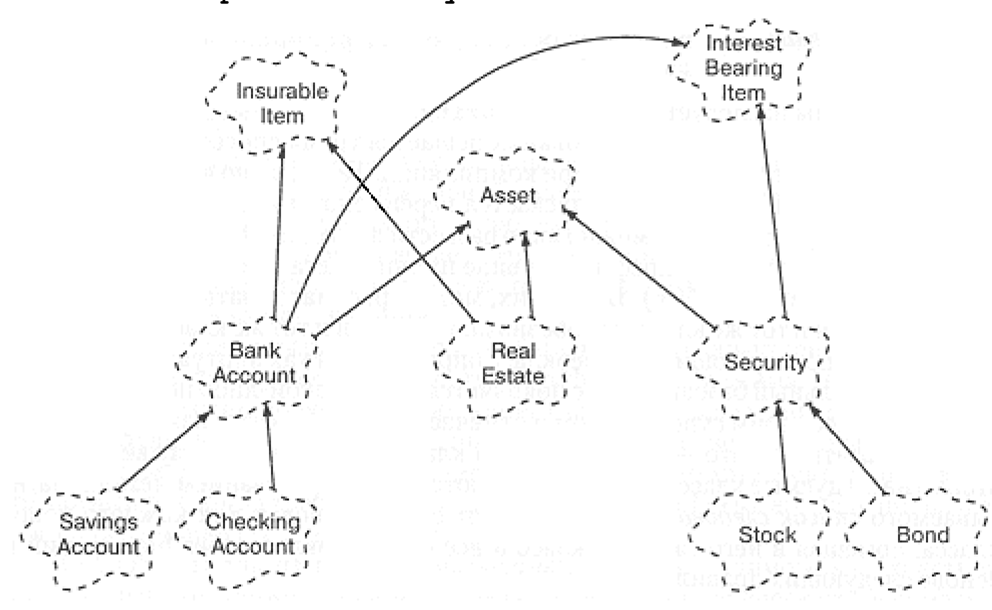


Рис. 3-7. Множественное наследование

Проектирование структур классов со множественным наследованием - трудная задача, решаемая путем последовательных приближений. Есть две специфические для множественного наследования проблемы - как разрешить конфликты имен между суперклассами и что делать с повторным наследованием.

Конфликт имен происходит, когда в двух или более суперклассах случайно оказывается элемент (переменная или метод) с одинаковым именем. Представьте себе, что как **Asset**, так и **InsurableItem** содержат атрибут **presentValue**, обозначающий текущую стоимость. Так как класс **RealEstate** наследует обоим этим классам, как понимать наследование двух операций с одним и тем же именем? Это, на самом деле, главная беда множественного наследования: конфликт имен может ввести двусмысленность в поведение класса с несколькими предками.

Борются с этим конфликтом тремя способами. Во-первых, можно считать конфликт имен ошибкой и отвергать его при компиляции (так делают Smalltalk и Eiffel, хотя в Eiffel конфликт можно разрешить, исправив имя). Во-вторых, можно считать, что одинаковые имена означают одинаковый атрибут (так делает CLOS). В третьих, для устранения конфликта разрешается добавить к именам префиксы, указывающие имена классов, откуда они пришли. Такой подход принят в C++.[16]

О второй проблеме, повторном наследовании, Мейер пишет следующее: "Одно тонкое затруднение при использовании множественного наследования встречается, когда один класс является наследником другого по нескольким линиям. Если в языке разрешено множественное наследование, рано или поздно кто-нибудь напишет класс D, который наследует от в и с, которые, в свою очередь, наследуют от A. Эта ситуация называется повторным наследованием, и с ней нужно корректно обращаться" [41]. Рассмотрим следующий класс:

```
class MutualFund :   public Stock,
                         public Bond ...
```

который дважды наследует от класса **security**.

Проблема повторного наследования решается тремя способами. Во-первых, можно его запретить, отслеживая при компиляции. Так сделано в языках Smalltalk и Eiffel (но в Eiffel, опять-таки допускается переименование для устранения неопределенности). Во-вторых, можно явно развести две копии унаследованного элемента, добавляя к именам префиксы в виде имени класса-источника (это один из подходов, принятых в C++). В-третьих, можно рассматривать множественные ссылки на один и тот же класс, как обозначающие один и тот же класс. Так поступают в C++, где повторяющийся суперкласс определяется как виртуальный базовый класс. Виртуальный базовый класс появляется, когда какой-либо подкласс именует другой класс своим суперклассом и отмечает этот суперкласс как виртуальный, чтобы показать, что это - общий (shared) класс. Аналогично, в языке CLOS повторно наследуемые классы "обобществляются" с использованием механизма, называемого список следования классов. Этот список заводят для каждого нового класса, помещая в него сам этот класс и все его суперклассы без повторений на основе следующих правил:

• класс всегда предшествует своему суперклассу;

• каждый класс сам определяет порядок следования своих непосредственных родителей.

В результате граф наследования оказывается плоским, дублирование устраняется, и появляется возможность рассматривать результирующую иерархию как иерархию с одиночным наследованием [43]. Это весьма напоминает топологическую сортировку классов. Если она возможна, то повторное наследование допускается. При этом теоретически могут существовать несколько равноправных результатов сортировки, но алгоритм так или иначе выдает какой-то один из них. Если же сортировка невозможна (например, в структуре возникают циклы), то класс отвергается.

При множественном наследовании часто используется прием создания примесей (**mixin**). Идея примесей происходит из языка Flavors: можно комбинировать (смешивать) небольшие классы, чтобы строить классы с более сложным поведением. Хендлер пишет об этом так: "примесь синтаксически ничем не отличается от класса, но назначение их разное. Примесь не предназначена для порождения самостоятельно используемых экземпляров - она смешивается с другими классами" [44]. На рис. 3-7 классы **Insurableltem** и **interestBearingltem** - это примеси. Ни один из них не

---

[16] В C++ конфликт имен элементов подкласса может быть разрешен полной квалификацией имени члена класса. Функции-члены с одинаковыми именами и сигнатурами семантическими считаются идентичными.

может существовать сам по себе, они используются для придания смысла другим классам.[17] Таким образом, примесь - это класс, выражающий не поведение, а одну какую-то хорошо определенную повадку, которую можно привить другим классам через наследование. При этом повадка эта обычно ортогональна собственному поведению наследующего ее класса. Классы, сконструированные целиком из примесей, называют агрегатными.

Множественный полиморфизм. Вернемся к одной из функций-членов класса **DisplayIem**:

```
virtual void draw();
```

Эта операция изображает объект на экране в некотором контексте. Она объявлена виртуальной, то есть полиморфной, переопределяемой подклассами. Когда эту операцию вызывают для какого-то объекта, программа определяет, что, собственно, выполнять (см. врезку выше). Это одиночный полиморфизм в том смысле, что смысл сообщения зависит только от одного параметра, а именно, объекта, для которого вызывается операция.

На самом деле операция **draw** должна бы зависеть от характеристик используемой системы отображения, в частности от графического режима. Например, в одном случае мы хотим получить изображение с высоким разрешением, а в другом - быстро получить черновое изображение. Можно ввести две различных операции, скажем, **drawGraphic** и **drawText**, но это не совсем то, что хотелось бы. Дело в том, что каждый раз, когда требуется учесть новый вид устройства, его надо проводить по всей иерархии надклассов для класса **Displayltem**.

В CLOS есть так называемые мультиметоды. Они полиморфны, то есть их смысл зависит от множества параметров (например, от графического режима и от объекта). В C++ мультиметодов нет, поэтому там используется идиома так называемый двойной диспетчеризации.

Например, мы могли бы вести иерархию устройств отображения информации от базового класса **DisplayDevice**, а атем определить метод класса **Displayltem** так:

```
virtual void draw(DisplayDevice&);
```

При реализации этого метода мы вызываем графические операции, которые полиморфны относительно переданного параметра типа **Displayltem**, таким образом происходит двойная диспетчеризация: **draw** сначала демонстрирует полиморфное поведение в зависимости от того, к какому подклассу класса **Displayltem** принадлежит объект, а затем полиморфизм проявляется в зависимости от того, к какому подклассу класса **DisplayDevice** принадлежит аргумент. Эту идиому можно продолжить до множественной диспетчеризации.

Агрегация

Пример. Отношение агрегации между классами имеет непосредственное отношение к агрегации между их экземплярами. Рассмотрим вновь класс **TemperatureController**:

```
class TemperatureController {
public:
        TemperatureController(Location) ;
        ~TemratureController();
        void process(const TemperatureRamp&);
        Minute schedule(const TemperatureRamp&) const;
```

[17] Для языка CLOS при обогащении поведения существующих первичных методов обычной практикой является строить примесь, используя только : before- и : after-методы.

```
private:
      Heater h;
};
```

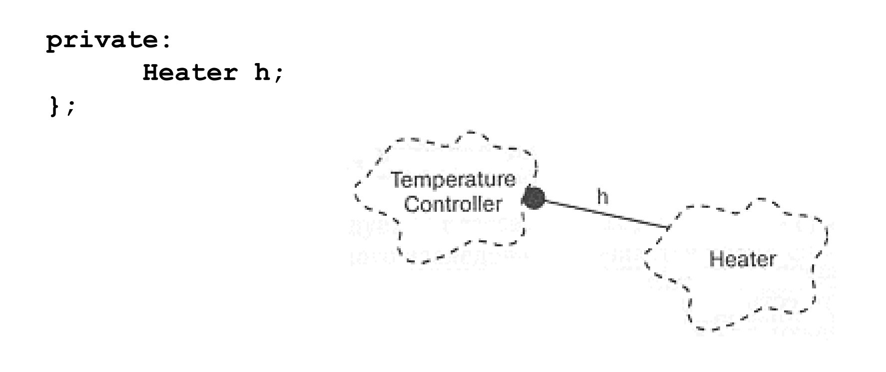


Рис. 3-8. Агрегация

Как явствует из рис. 3-8, класс **TemperatureController** это, несомненно, целое, а экземпляр класса Heater - одна из его частей. Совершенно такое же отношение агрегации между экземплярами этих классов показано на рис. 3-3.

Физическое включение. В случае класса **TemperatureController** мы имеем агрегацию по значению; эта разновидность физического включения означает, что объект класса **Heater** не существует отдельно от объемлющего экземпляра класса **TemperatureController**.

Менее обязывающим является включение по ссылке. Мы могли бы изменить закрытую часть **TemperatureController** так:[18]

```
Heater* h;
```

В этом случае класс **TemperatureController** по-прежнему означает целое, но его часть, экземпляр класса Heater, содержится в целом косвенно. Теперь эти объекты живут отдельно друг от друга: мы можем создавать и уничтожать экземпляры классов независимо. Чтобы избежать структурной зависимости через ссылки важно придерживаться какой-то договоренности относительно создания и уничтожения объектов, ссылки на которые могут содержаться в разных местах. Нужно, чтобы это делал кто-то один.

Агрегация является направленной, как и всякое отношение "целое/часть". Объект Heater входит в объект TemperatureController, и не наоборот. Физическое вхождение одного в другое нельзя "зациклить", а вот указатели - можно (каждый из двух объектов может содержать указатель на другой).[19]

Конечно, как уже говорилось, агрегация не требует обязательного физического включения, ни по значению, ни по ссылке. Например, акционер владеет акциями, но они не являются его физической частью. Более того, время жизни этих объектов может быть совершенно различным, хотя концептуально отношение целого и части сохраняется и каждая акция входит в имущество своего акционера. Поэтому агрегация может быть очень косвенной. Например, объект класса **Shareholder** (акционер) может содержать ключ записи об этом акционере в базе данных акций. Это тоже агрегация без физического включения. "Лакмусовая бумажка" для выявления агрегации такова: если (и только если) налицо отношение

---

[18] В качестве альтернативы мы могли бы описать h как ссылку на нагреватель (Heater& в C++), в этом случае семантика инициализации и модификации этого объекта будет совершенно отличной от семантики указателей.

[19] Ассоциация часто может быть заменена циклической агрегацией или циклическими отношениями использования. Однако чаще ассоциация, которая по определению предполагает двунаправленность, во время проектирования превращается в простую агрегацию или отношение использования, выявляя ограничения на направление ассоциации.

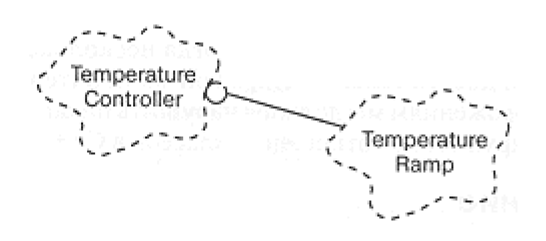


Рис. 3-9. Отношение использования

"целое/часть" между объектами, их классы должны находиться в отношении агрегации друг с другом.

Часто агрегацию путают с множественным наследованием. Действительно, в C++ скрытое (защищенное или закрытое) наследование почти всегда можно заменить скрытой агрегацией экземпляра суперкласса. Решая, с чем вы имеете дело - с наследованием или агрегацией - будьте осторожны. Если вы не уверены, что налицо отношение общего и частного (**is** a), вместо наследования лучше применить агрегацию или что-нибудь еще.

Использование

Пример. В недавнем примере объекты **rampController** и **growingRamp** иллюстрировали связь между объектами, которую мы представляли в виде отношения использования между их классами **TemperatureController** и **TemperatureRamp**.

```
class TemperatureController {
public:
      TemperatureController(Location) ;
      ~TemperatureController();
      void process(const TemperatureRamp&);
      Minute schedule(const TemperatureRamp&) const;
private:
      Heater h;
};
```

Класс **TemperatureRamp** упомянут как часть сигнатуры функции-члена process;

это дает нам основания сказать, что класс **TemperatureController** пользуется услугами класса **TemperatureRamp**.

Клиенты и серверы. Отношение использования между классами соответствует равноправной связи между их экземплярами. Это то, во что превращается ассоциация, если оказывается, что одна из ее сторон (клиент) пользуется услугами другой (сервера). Пример клиент-серверных отношений показан на рис. 3-9.

На самом деле, один класс может использовать другой по-разному. В нашем примере это происходит в сигнатуре интерфейсной функции. Можно представить, что **TemperatureController** внутри реализации функции **schedule** использует, например, экземпляр класса **Predictor** (предсказатель). Отношения целого и части тут ни при чем, поскольку этот объект не входит в объект **TemperatureController**, а только используется. В типичном случае такое отношение использования проявляет себя, если в реализации какой-либо операции происходит объявление локального объекта используемого класса.

Строгое отношение использования иногда несколько ограничительно, поскольку клиент имеет доступ только к открытой части интерфейса сервера.

Иногда по тактическим соображениям мы должны нарушить инкапсуляцию, для чего, собственно, и служат "дружеские" отношения классов в С++.

Инстанцирование

Примеры. Наша первая попытка сконструировать класс Queue (очередь) была не особенно успешной, поскольку нам не удалось сделать его безопасным в отношении типов. Мы можем значительно усовершенствовать нашу абстракцию, если прибегнем к конструкции параметризованных классов, которая поддерживается языками C++ и Eiffel.

```
Template<class Item>
class Queue {
public:
      Queue() ;
      Queue(const Queue<Item>&);
      virtual ~Queue();
      virtual Queue<Item>& operator=(const Queue<Item>&);
      virtual int operator==(const Queue<Item>&) const;
      int operator!=(const Queue<Item>&) const;
      virtual void clear();
      virtual void append(const Item&);
      virtual void pop();
      virtual void remove(int at);
      virtual int length() const;
      virtual int isEmpty() const;
      virtual const Item& front() const;
      virtual int location(const void*);
protected:
…
};
```

В этом новом варианте не используется идиома **void***, вместо этого объекты помещаются в очередь и достаются из нее через класс **item**, объявленный как аргумент шаблона.

Параметризованный класс не может иметь экземпляров, пока он не будет ин-станцирован. Объявим две конкретных очереди - очередь целых чисел и очередь экранных объектов:

```
      Queue<int> intQueue;
      Queue<DisplayItem*> itemQueue;
```

Объекты intQueue и **itemQueue** - это экземпляры совершенно различных классов, которые даже не имеют общего суперкласса. Тем не менее, они получены из одного параметризованного класса **Queue**. По причинам, которые мы объясним позже в главе 9, во втором случае мы поместили в очередь указатели. Благодаря этому, любые объекты подклассов **DisplayItem**, помещенные в очередь, не будут "срезаться", но сохранят свое полиморфное поведение.

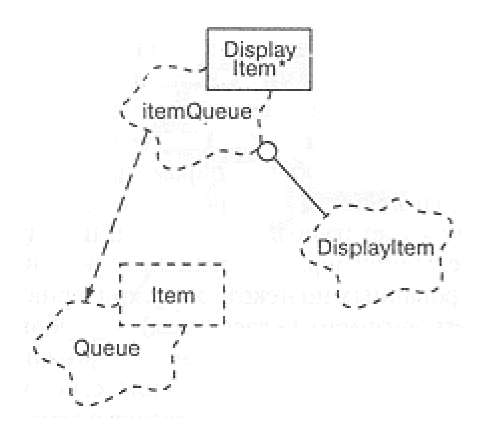


Рис. 3-10. Инстанцирование

Это Инстанцирование безопасно с точки зрения типов. По правилам C++ будет отвергнута любая попытка поместить в очередь или извлечь из нее что-либо кроме, соответственно, целых чисел и разновидностей `DisplayItem`.

Отношения между параметризованным классом `Queue`, его инстанцированием для класса `DisplayItem` и экземпляром `itemQueue` показаны на рис. 3-10.

Обобщенные классы. Существует четыре основных способа создавать такие классы, как параметризованный класс `Queue`. Во-первых, мы можем использовать макроопределения. Именно так это было в раннем C++, но, как пишет Страуструп, "данный подход годился только для небольших проектов" [45], так как макросы неуклюжи и находятся вне семантики языка, более того, при каждом инстанцировании создается новая копия программного кода. Во-вторых, можно положиться на позднее связывание и наследование, как это делается в Smalltalk [46]. При таком подходе мы можем строить только неоднородные контейнерные классы, так как в языке нет средства ввести нужный класс элементов контейнера; каждый элемент в контейнере трактуется как экземпляр некоторого удаленного базового класса. Третий способ реализован в языках семейства Object Pascal, которые имеют и сильные типы, и наследование, но не поддерживают никакой разновидности параметризованных классов. В этом случае приходится создавать обобщенные контейнеры, как в Smalltalk, но использовать явную проверку типа объекта, прежде чем помещать его в контейнер. Наконец, есть собственно параметризованные классы, впервые появившиеся в CLU. Параметризованный класс представляет собой что-то вроде шаблона для построения других классов; шаблон может быть параметризован другими классами, объектами или операциями. Параметризованный класс должен быть инстанцирован перед созданием экземпляров. Механизм обобщенных классов есть в C++ и Eiffel.

Как можно заметить из рис. 3-10, чтобы инстанцировать параметризованный класс Queue мы должны использовать другой класс, например, DisplayItem. Благодаря этому отношение инстанцирования почти всегда подразумевает отношение использования.

Мейер указывает, что наследование - более мощный механизм, чем обобщенные классы и что через наследование можно получить большинство преимуществ обобщенных классов, но не наоборот [47]. Нам кажется, что лучше, когда языки поддерживают и то, и другое.

Параметризованные классы полезны далеко не только для создания контейнеров. Например, Страуструп отмечает их значение для обобщенной арифметики [48].

При проектировании обобщенные классы позволяют выразить некоторые свойства протоколов классов. Класс экспортирует операции, которые можно выполнять над его экземплярами. Наоборот, параметризующий аргумент класса служит для импорта классов и значений,

предоставляющих некоторый протокол. C++ проверяет их взаимное соответствие при компиляции, когда фактически и происходит инстанцирование. Например, мы могли бы определить упорядоченную очередь объектов, отсортированных по некоторому критерию. Этот параметризованный класс должен иметь аргумент (класс **I**tem), и требовать от этого аргумента определенное поведение (наличие операции вычисления порядка). При инстанци-ровании в качестве класса **I**tem годится любой класс, который имеет соответствующий протокол. Таким образом, поведение классов в семействе, происходящем от одного параметризованного класса, может изменяться в весьма широких пределах.

Метаклассы

Как было сказано, любой объект является экземпляром какого-либо класса. Что будет, если мы попробуем и с самими классами обращаться как с объектами? Для этого нам надо ответить на вопрос, что же такое класс класса? Ответ - это метакласс. Иными словами, метакласс - это класс, экземпляры которого суть классы. Метаклассы венчают объектную модель в чисто объектно-ориентированных языках. Соответственно, они есть в Smalltalk и CLOS, но не в C++.

Вот как Робсон мотивирует потребность в метаклассах: "классы доставляют программисту интерфейс для определения объектов. Если так, то желательно, чтобы и сами классы были объектами, так, чтобы ими можно было манипулировать, как всеми остальными описаниями" [49].

В языках типа Smalltalk первичное назначение метакласса - поддержка переменных класса (которые являются общими для всех экземпляров этого класса), операции инициализации переменных класса и создания единичного экземпляра метакласса [50]. По соглашению, метакласс Smalltalk обычно содержит примеры использования его классов. Например, как показано на рис. 3-11, мы могли бы задать переменную класса **nextId** для метакласса **TelemetryData**, чтобы вырабатывать идентифицирующие метки при создании каждого экземпляра **TelemetryData**. Аналогично, мы могли бы определить оператор порождения новых экземпляров класса, который изготавливал бы их, скажем, в некотором предварительно выделенном пуле памяти.

Хотя в C++ метаклассов нет, семантика его конструкторов и деструкторов служит целям, аналогичным тем, что вызвали к жизни метаклассы. C++ имеет средства поддержки и переменных класса, и операций метакласса. Конкретно, в C++ можно описать члены данных или функции класса как статические (**static**), что будет означать: этот элемент является общим для всех экземпляров класса. Статические члены класса в C++ эквивалентны переменным класса в Smalltalk. Статическая функция-член класса играет роль операций метакласса в Smalltalk.

Как мы уже отмечали, в CLOS аппарат метаклассов еще сильнее чем в Smalltalk. Через него можно изменять саму семантику элементов: следование классов, обобщенные функции и методы. Главное преимущество - возможность экспериментировать с другими объектно-ориентированными парадигмами и создавать такие инструменты для разработчика, как броузеры классов и объектов.

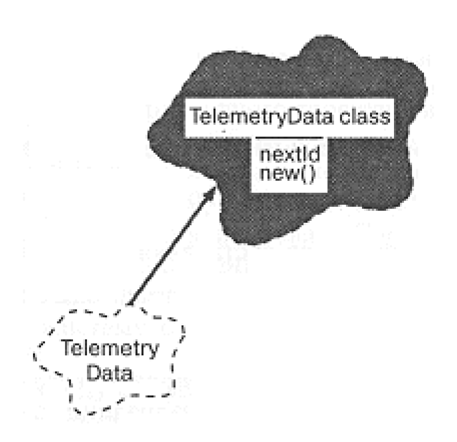


Рис. 3-11. Метаклассы

В CLOS есть предопределенный класс с именем standard-class, который является метаклассом для всех нетипизированных классов, определенных с помощью defclass. В этом метаклассе есть метод make-instance, который создает экземпляры. Кроме того, в нем определена вся техника работы со списком следования классов. Все это можно изменить.

Методы и обобщенные функции в CLOS тоже можно рассматривать как объекты. Так как они несколько отличаются от обычных объектов, то в совокупности объекты, соответствующие классам, методам и обобщенным функциям, называются *метаобьектами*. Каждый метод является экземпляром предопределенного класса **standard-method**, а каждая функция является экземпляром предопределенного класса **standard-generic-function.** Поскольку поведение этих предопределенных классов можно изменить, удается влиять на трактовку методов и обобщенных функций.

3.5. Взаимосвязь классов и объектов

Отношения между классами и объектами

Классы и объекты - это отдельные, но тесно связанные понятия. В частности, каждый объект является экземпляром какого-либо класса; класс может порождать любое число объектов. В большинстве практических случаев классы статичны, то есть все их особенности и содержание определены в процессе компиляции программы. Из этого следует, что любой созданный объект относится к строго фиксированному классу. Сами объекты, напротив, в процессе выполнения программы создаются и уничтожаются.

В качестве примера рассмотрим классы и объекты для задачи управления воздушным движением. Наиболее важные абстракции в этой сфере - самолеты, графики полетов, маршрут и коридоры в воздушном пространстве. Трактовка этих классов объектов по самому их определению достаточно статична. Иначе невозможно было бы построить никакого приложения, использующего такие общепонятные факты, как то, что самолеты могут взлетать, летать и приземляться, а также что никакие два самолета не должны находиться одновременно в одной и той же точке.

Объекты же этих классов, напротив, динамичны. Набор маршрутов полетов сменяется не очень часто. Существенно быстрее изменяется множество самолетов, находящихся в полете. Частота, с которой самолеты занимают и покидают воздушные коридоры, еще выше.

Роль классов и объектов в анализе и проектировании

На этапе анализа и ранних стадиях проектирования решаются две основные задачи:

• Выявление классов и объектов, составляющих словарь предметной области.

• Построение структур, обеспечивающих взаимодействие объектов, при котором выполняются требования задачи.

В первом случае говорят о ключевых абстракциях задачи (совокупность классов и объектов), во втором - о механизмах реализации (совокупность структур).

На ранних стадиях внимание проектировщика сосредоточивается на внешних проявлениях ключевых абстракций и механизмов. Такой подход создает логический каркас системы: структуры классов и объектов. На последующих фазах проекта, включая реализацию, внимание переключается на внутреннее поведение ключевых абстракций и механизмов, а также их физическое представление. Принимаемые в процессе проектирования решения задают архитектуру системы: и архитектуру процессов, и архитектуру модулей.

3.6. Качество классов и объектов

Измерение качества абстракции

По мнению Ингалса "для построения системы должен использоваться минимальный набор неизменяемых компонент; сами компоненты должны быть по возможности стандартизованы и рассматриваться в рамках единой модели" [51]. Применительно к объектно-ориентированному проектированию такими компонентами являются классы и объекты, отражающие ключевые абстракции системы, а единство обеспечивается соответствующими механизмами реализации.

Опыт показывает, что процесс выделения классов и объектов является последовательным, итеративным. За исключением самых простых задач с первого раза не удается окончательно выделить и описать классы. В главах 4 и 7 показано, как в процессе работы сглаживаются противоречия, возникающие при начальном определении абстракций. Очевидно, такой процесс связан с дополнительными затратами на перекомпиляцию, согласование и внесение изменений в проект системы. Очень важно, следовательно, с самого начала по возможности приблизиться к правильным решениям, чтобы сократить число последующих шагов приближения к истине. Для оценки качества классов и объектов, выделяемых в системе, можно предложить следующие пять критериев:

• зацепление
• связность
• достаточность
• полнота
• примитивность.

Термин *зацепление* взят из структурного проектирования, но в более вольном толковании он используется и в объектно-ориентированном проектировании. Стивенс, Майерс и Константайн определяют зацепление как "степень глубины связей между отдельными модулями. Систему с сильной зависимостью между модулями гораздо сложнее воспринимать и модифицировать. Сложность системы может быть уменьшена путем уменьшения зацепления между отдельными модулями" [52]. Пример неправильного подхода к проблеме зацепления привел Пейдж-Джонс, описав модульную стереосистему, в которой источник питания размещен в одной из звуковых колонок [53].

Кроме зацепления между модулями в объектно-ориентированном анализе, существенно зацепление между классами и объектами. Существует определенное противоречие между явлениями зацепления и наследования. С

одной стороны, желательно избегать сильного зацепления классов; с другой стороны, механизм наследования, тесно связывающий подклассы с суперклассами, помогает выгодно использовать сходство абстракций.

Понятие связности также заимствовано из структурного проектирования. Связность - это степень взаимодействия между элементами отдельного модуля (а для OOD еще и отдельного класса или объекта), характеристика его насыщенности. Наименее желательной является связность по случайному принципу, когда в одном классе или модуле собираются совершенно независимые абстракции. Для примера можно вообразить класс, соединяющий абстракции собак и космических аппаратов. Наиболее желательной является функциональная связность, при которой все элементы класса или модуля тесно взаимодействуют в достижении определенной цели. Так, например, класс Dog будет функционально связным, если он описывает поведение собаки, всей собаки, и ничего, кроме собаки.

К идеям зацепления и связности тесно примыкают понятия достаточности, полноты и примитивности. Под достаточностью подразумевается наличие в классе или модуле всего необходимого для реализации логичного и эффективного поведения. Иначе говоря, компоненты должны быть полностью пригодны к использованию. Для примера рассмотрим класс set (множество). Операция удаления элемента из множества в этом классе, очевидно, необходима, но будет ошибкой не включить в этот класс и операцию добавления элемента. Нарушение требования достаточности обнаруживается очень быстро, как только создается клиент, использующий абстракцию. Под полнотой подразумевается наличие в интерфейсной части класса всех характеристик абстракции. Идея достаточности предъявляет к интерфейсу минимальные требования, а идея полноты охватывает все аспекты абстракции. Полнотой характеризуется такой класс или модуль, интерфейс которого гарантирует все для взаимодействия с пользователями. Полнота является субъективным фактором, и разработчики часто ею злоупотребляют, вынося на верх такие операции, которые можно реализовать на более низком уровне. Из этого вытекает требование примитивности. Примитивными являются только такие операции, которые требуют доступа к внутренней реализации абстракции. Так, в примере с классом set операция Add (добавление к множеству элемента) примитивна, а операция добавления четырех элементов не будет примитивной, так как вполне эффективно реализуется через операцию добавления одного элемента. Конечно, эффективность тоже вещь субъективная. Операция, которая требует прямого доступа к структуре данных, примитивна по определению. Операция, которая может быть описана в терминах существующих примитивных операций, но ценой значительно больших вычислительных затрат, также является кандидатом на включение в разряд примитивных.[20]

Как выбрать операции?

Функциональность. Описание интерфейса класса или модуля - трудная работа. Обычно первое приближение делается, исходя из структурного смысла класса, а затем, когда появляются клиенты класса, интерфейс уточняется, модифицируется и дополняется. В частности может возникнуть потребность в создании новых классов или в изменении взаимодействия существующих.

В пределах каждого класса принято иметь только примитивные операции, отражающие отдельные аспекты поведения. Такие методы называются точными. Принято также отделять методы, не связанные между собой. Это облегчает образование подклассов с переопределением поведения. Решение о количестве методов может быть обусловлено двумя причинами:

---

[20] Примером может служить операция добавления к множеству произвольного числа элементов (а не обязательно четырех). - Примеч. ред.

описание поведения в одном методе упрощает интерфейс, но усложняет и увеличивает размеры самого метода; расщепление метода усложняет интерфейс, но делает каждый из методов проще. По наблюдению Мейера "хороший проектировщик умеет найти компромисс между большим числом связей (дробление системы на фрагменты) и большим размером модулей (что может привести к потере управляемости)" [54].

В объектно-ориентированном проектировании принято рассматривать методы класса как единое целое, поскольку все они взаимодействуют друг с другом для реализации протокола абстракции. Таким образом, определив поведение, нужно решить, в каком из классов это поведение реализуется. Халберт и 0'Брайен предложили следующие критерии для принятия такого решения:

• Повторная используемость   Будет ли это поведение полезно более чем в одном контексте?

• Сложность   Насколько трудно реализовать такое поведение?

• Применимость   Насколько данное поведение характерно для класса, в который мы хотим включить поведение?

• Знание реализации   Надо ли для реализации данного поведения знать секреты класса?

Обычно операции объявляются, как методы класса, к объектам которого относятся данные действия. Однако в языках Object Pascal, C++, CLOS и Ada допускается описание операций в виде свободных подпрограмм (утилит класса). Свободная подпрограмма, в терминологии C++, - это функция, не являющаяся элементом класса. Свободные подпрограммы не могут переопределяться подобно обычным методам, в них нет такой общности. Наличие утилит позволяет выполнить требование примитивности и уменьшить зацепление между классами, особенно если эти операции высокого уровня задействуют объекты многих различных классов.

Аспекты расхода памяти и времени. После того, как мы приняли решение о необходимости конкретной функции и определили ее семантику, следует принять решение об использовании ею времени и памяти. Для выражения таких решений принято использовать понятие лучшего, среднего и худшего вариантов, где худший - это верхний допустимый предел расходов.

Раньше мы уже отмечали, что поскольку один объект посылает другому сообщение, эти два объекта должны быть каким-то образом синхронизированы. В случае многих потоков управления это означает, что передача сообщений сложнее, чем управление вызовами подпрограмм. Для большинства языков программирования синхронизация просто не нужна, поскольку в них программы однопотоковые, и все объекты действуют последовательно. Мы говорим в таких случаях о простой передаче сообщений, так как ее семантика больше похожа на простой вызов подпрограмм. Однако в языках, поддерживающих параллелизм,[21] нужно побеспокоиться о более изощренных системах передачи сообщений, чтобы избежать случаев, когда два потока работают одновременно и несогласованно с одним и тем же объектом. Объекты, семантика которых сохраняется при многопоточности, являются или синхронизированными, или защищенными.

В некоторых обстоятельствах полезно отмечать параллельность как для отдельных операций, так и для объекта в целом, так как разные операции могут потребовать разных форм синхронизации. Выделяют следующие формы передачи сообщений:

---

[21] Ada и Smalltalk имеют прямую поддержку параллельности. Языки типа C++ такой поддержкой не обладают, но в них часто можно обеспечить семантику параллельности за счет расширения классами (зависящими от платформы): примером служит библиотека AT&T для C++.

• Синхронная Операция активизируется только при готовности передающего и принимающего сообщения объектов; ожидание взаимной готовности может быть неопределенно долгим

• С учетом задержки То же, что и синхронная, однако, в случае, если принимающий не готов, передающий не выполняет операцию

• С ограничением времени То же, что и синхронная, однако, посылающий будет ждать готовности принимающего не дольше некоторого времени

• Асинхронная Операция выполняется вне зависимости от готовности принимающего.

Нужная форма выбирается для каждой операции отдельно, но только после того, как ее функциональная семантика определена.

Как выбирать отношения

Сотрудничество. Отношения между классами и объектами связаны с конкретными действиями. Если мы хотим, чтобы объект X послал объекту Y сообщение M, то прямо или косвенно класс x должен иметь доступ к классу Y, иначе невозможно вызвать в классе x операцию м. Под доступностью мы понимаем способность одной абстракции видеть другую и обращаться к ее открытым ресурсам. Абстракции доступны одна другой только тогда, когда перекрываются их области видимости и даны необходимые права доступа (так, закрытая часть класса доступна только ему самому и его друзьям). Таким образом, зацепление связано с видимостью.

Одним из полезных правил является закон Деметера, который утверждает, что "методы любого класса не должны зависеть от структуры других классов, а только от структуры (верхнего уровня) самого класса. В каждом методе посылаются сообщения только объектам из предельно ограниченного множества классов" [56]. Следование этому закону позволяет создавать слабо зацепленные классы, реализация которых скрыта. Такие классы достаточно автономны и для понимания их логики нет необходимости знать строение других классов.

При анализе структуры классов системы в целом можно обнаружить, что иерархия наследования либо широкая и мелкая, либо узкая и глубокая, либо сбалансированная. В первом случае структура классов выглядит как лес из свободно стоящих деревьев. Классы могут свободно смешиваться и вступать во взаимоотношения [57]. Во втором случае структура классов напоминает одно дерево с ветвями классов, имеющих общего предка [58]. Каждый из вариантов имеет свои достоинства и недостатки. Классы, составляющие лес, независимы друг от друга, но, вероятно, не лучшим образом используют возможности специализации и обобществления кода. В случае дерева классов эта "коммунальность" используется максимально, поэтому каждый из классов имеет меньший размер. Однако в последнем случае классы невозможно понять без контекста всех их предков.

Иногда требуется выбирать между отношениями наследования, агрегации и использования. Например, должен ли класс Car (автомобиль) наследовать, содержать или использовать классы Engine (двигатель) и wheel (колесо)? В данном случае более целесообразны отношения использования. По мнению Мейера, между классами А И В "отношения наследования более пригодны тогда, когда любой объект класса в может одновременно рассматриваться и как объект А" [59]. С другой стороны, если объект является чем-то большим, чем сумма его частей, то отношение агрегации не совсем уместно.

Механизмы и видимость. Отношения между объектами определяется в основном механизмами их взаимодействия. Вопрос состоит только в том, кто о чем должен знать. Например, на ткацкой фабрике материалы (партии) поступают на участки для обработки. Как только они попадают на участок, об

этом надо известить управляющего. Является ли поступление материала на участок операцией над участком, над материалом, или тем и другим сразу? Если это операция над участком, то класс участка должен быть видим для материала. Если это операция над материалом, то класс материала должен быть видим для участка, так как партия материала должна различать участки. В случае операции над помещением и участком нужно обеспечить взаимную видимость. Аналогично следует определить отношение между управляющим и участком (но не материалом и управляющим): либо управляющий должен знать об участке, либо участок об управляющем.

Иногда в процессе проектирования полезно явно определить видимость объектов. Существуют четыре основных способа сделать так, чтобы объект X (клиент) видел объект Y (сервер):

• сервер является глобальным;

• сервер передается клиенту в качестве параметра операции;

• сервер является частью клиента в смысле классов;

• сервер локально объявляется в области видимости клиента.

Эти варианты можно комбинировать. Y может быть частью x и при этом быть видимым другим объектам. В языке Smalltalk такой способ обычно означает зависимость между двумя объектами. Общая зона видимости приводит к структурной зависимости, то есть один объект не имеет исключительных прав доступа к другому: состояние этого другого объекта может быть изменено несколькими способами.

Выбор реализации

Внутреннее строение (реализация) классов и объектов разрабатывается только после завершения проектирования их внешнего поведения. При этом необходимо принять два проектных решения: выбрать способ представления класса или объекта и способ размещения их в модуле.

Представление. Представление классов и объектов почти всегда должно быть инкапсулировано (скрыто). Это позволяет вносить изменения (например, перераспределение памяти и временных ресурсов) без нарушения функциональных связей с другими классами и объектами. Как мудро отметил Вирт: "выбор способа представления является нелегкой задачей и не определяется одними лишь техническими средствами. Он всегда должен рассматриваться с точки зрения операций над данными" [60]. Рассмотрим, например, класс, соответствующий расписаниям полетов самолетов. Как его нужно оптимизировать - по эффективности поиска или по скорости добавления/удаления рейса? Поскольку невозможно реализовать и то, и другое одновременно, нужно сделать выбор, исходя из целей системы. Иногда такой выбор сделать непросто, и тогда создается семейство классов с одинаковым интерфейсом, но с принципиально разной реализацией для обеспечения вариативности поведения.

Одним из наиболее трудных решений является выбор между вычислением элементов состояния объекта и хранением их в виде полей данных. Рассмотрим, например, класс Cone (конус) с соответствующим ему методом volume (объем). Этот метод возвращает значение объема объекта. В структуре конуса в виде отдельных полей хранятся данные о его высоте и радиусе основания. Следует ли еще создать поле данных для объема или следует вычислять его по мере необходимости внутри метода volume [60]? Если мы хотим получать значение объема максимально быстро, нужно создавать соответствующее поле данных. Если важнее экономия памяти, лучше вычислить это значение. Оптимальный способ представления объекта всегда определяется характером решаемой задачи. В любом случае этот выбор не должен влиять на интерфейс класса.

Модульная структура. Аналогичные вопросы возникают при распределении деклараций классов и объектов по модулям. В языке Smalltalk

эта проблема отсутствует, здесь модульный механизм не реализован. В языках Object Pascal, C++, CLOS и Ada существует понятие модуля как отдельной языковой конструкции. Решение о месте декларирования классов и объектов в этих языках является компромиссом между требованиями видимости и скрытия информации. В общем случае модули должны быть функционально связными внутри и слабо связанными друг с другом. При этом следует учитывать ряд нетехнических факторов, таких, как повторное использование, безопасность, документирование. Проектирование модулей - не более простой процесс, чем проектирование классов и объектов. О скрытии информации Парнас, Клеменс и Вейс говорят следующее: "Применение этого принципа не всегда очевидно. Принцип нацелен на минимизацию стоимости программных средств (в целом за время эксплуатации), для чего от проектировщика требуется способность оценивать вероятность изменений. Такие оценки основываются на практическом опыте и знаниях предметной области, включая понимание технологии программирования и аппаратных особенностей" [61].

Выводы

- Объект характеризуется состоянием, поведением и идентичностью.
- Структура и поведение одинаковых объектов описывается в общем для них классе.
- Состояние объекта определяет его статические и динамические свойства.
- Поведение объекта характеризуется изменением его состояния в процессе взаимодействия (посредством передачи сообщений) с другими объектами.
- Идентичность объекта - это его отличия от всех других объектов.
- Иерархия объектов может строиться на принципах связи или агрегации.
- Множество объектов с одинаковой структурой и поведением является классом.
- Шесть типов иерархий классов включают: ассоциирование, наследование, агрегация, использование, инстанцирование и метаклассирование.
- Классы и объекты, образующие словарь предметной области, называются ключевыми абстракциями.
- Структура, объединяющая множество объектов и обеспечивающая их совместное целенаправленное функционирование, называется механизмом.
- Качество абстракций измеряется их зацеплением, связностью, достаточностью, полнотой и примитивностью.

Дополнительная литература

МакЛеннан (MacLennan) [G 1982] обсуждал различие между значениями и объектами. Работа Меиера (Meyer) Ц 1987] предлагает контрактный подход к программированию.

По поводу иерархии классов было написано много, особое внимание уделялось наследованию и полиморфизму. Работы Альбано (Albano) [G 1983], Аллена (Allen) [A 1982], Брахмана (Brachman) (J 1983], Хайлперна и Нгуена (Hailpern and Nguyen) [G 1987], и Вегнера и Здоника (Wegner and Zdonik) [J 1988] создали блестящее теоретическое обоснование всех основных вопросов и концепций. Кук и Палсберг (Cook and Palsberg) U 1989] и Турецкий (Touretzky) [G 1986] дали формальное истолкование семантики наследования. Вирт (Wirth) [] 1987] предложил сходные решения для обобщенных

структурных типов в Oberon. Ингалс (Ingalls) [G 1986] дал полезное обсуждение вопроса множественного полиморфизма. Грогоно (Grogono) [G 1989] изучает взаимодействие полиморфизма и проверки типов, а Пондер и Бач (Ponder and Buch) [G 1992] предупреждают об опасностях безграничного полиморфизма. Практические рекомендации по эффективному использованию наследования предложили Мейер (Meyer) [G 1988] и Халберд и О'Брайан (Halberd and O'Brien) [G 1988]. ЛаЛонд и Пух (LaLonde and Pugh) U 1985] изучали задачи обучения эффективному использованию специализации и обобщения.

Природа ролей и обязанностей абстракции подробна рассмотрена в работе Рубина и Голд-берга (Rubin and Goldberg) [В 1992], а также Вирфс-Брока, Вилкерсона и Винера (Wirfs-Brock, Wilkerson and Wiener) [F 1990]. Качество классов рассматривал также Коад (Coad)[F1991].

Мейер (Meyer) [G 1986] изучал связи между обобщенными функциями и наследованием применительно к языку Eiffel. Страуструп (Stroustrup) [G 1988] предложил механизм параметризованных типов в С++. Протокол метаобъектов в CLOS описали в деталях Кишалец, Ривьерес и Бобров (Kiczales, Rivieres, and Bobrow) [G 1991].

Альтернативу иерархии, основанной на классах, предоставляет делегирование, использующее только экземпляры. Этот подход детально рассмотрел Стейн (Stein) [G 1987].

Глава 4

Классификация

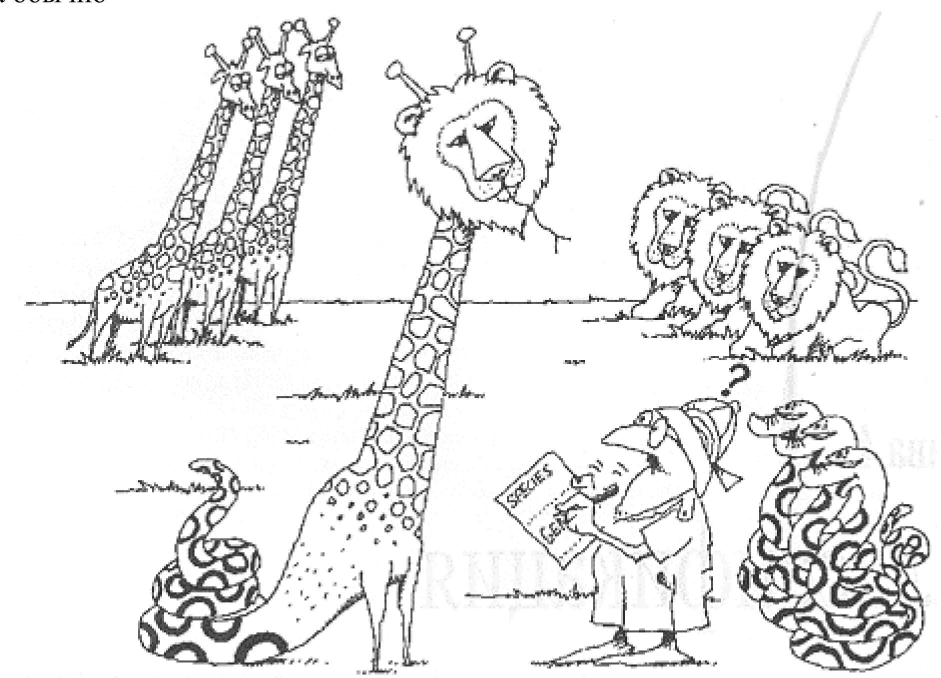
Классификация - средство упорядочения знаний. В объектно-ориентированном анализе определение общих свойств объектов помогает найти общие ключевые абстракции и механизмы, что в свою очередь приводит нас к более простой архитектуре системы. К сожалению, пока не разработаны строгие методы классификации и нет правила, позволяющего выделять классы и объекты. Нет таких понятий, как "совершенная структура классов", "правильный выбор объектов". Как и во многих технических дисциплинах, выбор классов является компромиссным решением.

На одной из конференций программистам был задан вопрос: "Какими правилами вы руководствуетесь при определении классов и объектов?" Страуструп, разработчик языка C++, ответил: "Это как поиск святого Грааля. Не существует панацеи". Габриель, один из разработчиков CLOS, сказал: "Это вопрос, на который нет простого ответа. Я просто пробую" [1]. К счастью, имеется богатый опыт классификации в других науках, на основе которого разработаны методики объектно-ориентированного анализа. Каждая такая методика предлагает свои правила (эвристики) идентификации классов и объектов. Они и будут предметом этой главы.

4.1. Важность правильной классификации

Классификация и объектно-ориентированное проектирование

Определение классов и объектов - одна из самых сложных задач объектно-ориентированного проектирования. Наш опыт показывает, что эта работа обычно



содержит в себе элементы открытия и изобретения. С помощью открытий мы распознаем ключевые понятия и механизмы, которые образуют словарь предметной области. С помощью изобретения мы конструируем обобщенные понятия, а также новые механизмы, которые определяют правила взаимодействия объектов. Поэтому открытие и изобретение - неотъемлемые части успешной классификации. Целью классификации является нахождение общих свойств объектов. Классифицируя, мы объединяем в одну группу объекты, имеющие одинаковое строение или одинаковое поведение.

Разумная классификация, несомненно, - часть любой науки. Михальски и Степп утверждают: "неотъемлемой задачей науки является построение содержательной классификации наблюдаемых объектов или ситуаций. Такая классификация существенно облегчает понимание основной проблемы и дальнейшее развитие научной теории" [2]. Та же философия относится и к инженерному делу. В области строительной архитектуры и городского планирования, как отмечает Алексан-дер, для архитектора "его проектная деятельность, и скромная, и гигантская по размеру, управляется целиком образами, которые он держит в своем сознании в данный момент, и его способностью комбинировать эти образы при создании нового проекта" [3].

Неудивительно, что классификация затрагивает многие аспекты объектно-ориентированного проектирования. Она помогает определить иерархии обобщения, специализации и агрегации. Найдя общие формы взаимодействия объектов, мы вводим механизмы, которые станут фундаментом реализации нашего проекта. Классификация помогает правильно определить модульную структуру. Мы можем расположить объекты в одном или разных модулях, в зависимости от степени схожести объектов; зацепление и связность - всего лишь меры этой схожести.

Классификация играет большую роль при распределении процессов между процессорами. Мы направляем процессы на один процессор или на разные в зависимости от того, как эти процессы связаны друг с другом.

Трудности классификации

Примеры классификации. В главе 3 мы определили объект как нечто, имеющее четкие границы. На самом деле это не вполне так. Границы предметов часто неопределенны. Например, посмотрите на вашу ногу. Попытайтесь определить, где начинается и кончается колено. В разговорной речи трудно понять, почему именно эти звуки определяют слово, а не являются частью какого-то более длинного слова. Представьте себе, что вы проектируете текстовый редактор. Что считать классом - буквы или слова? Как понимать отдельные фразы, предложения, параграфы, документы? Как обращаться с произвольными, не обязательно осмысленными, блоками текста? Что делать с предложениями, абзацами и целыми документами - соответствуют ли такие классы нашей задаче?

То, что разумная классификация - трудная проблема, новостью не назовешь. И поскольку есть параллели с аналогичными трудностями в объектно-ориентированном проектировании, рассмотрим примеры классификации в двух других научных дисциплинах: биологии и химии.

Вплоть до XVIII века идея о возможности классификации живых организмов по степени сложности была господствующей. Мера сложности была субъективной, поэтому неудивительно, что человек оказался в списке на первом месте. В середине XVIII века шведский ботаник Карл Линней предложил более подробную таксономию для классификации организмов: он ввел понятия рода и вида. Век спустя Дарвин выдвинул теорию, по которой механизмом эволюции является естественный отбор и ныне существующие виды животных - продукт эволюции древних организмов. Теория Дарвина основывалась на разумной классификации видов. Как утверждает Дарвин, "натуралисты пытаются расположить виды, роды, семейства в каждом классе в то, что называется натуральной системой. Что подразумевается под этой системой? Некоторые авторы понимают некоторую простую схему, позволяющую расположить наиболее похожие живые организмы в один класс и различные - в разные классы" [4]. В современной биологии термин "классификация" обозначает "установление иерархической системы категорий на основе предположительно существующих естественных связей между организмами" [5]. Наиболее общее понятие в биологической таксономии - царство, затем, в порядке убывания общности: тип (отдел), класс, отряд

(порядок), семейство, род и, наконец, вид. Исторически сложилось так, что место каждого организма в иерархической системе определяется на основании внешнего и внутреннего строения тела и эволюционных связей. В современной классификации живых существ выделяются группы организмов, имеющих общую генетическую историю, то есть организмы, имеющие сходные ДНК, включаются в одну группу. Классификация по ДНК полезна, чтобы различить организмы, которые похожи внешне, но генетически сильно отличаются. По современным воззрениям дельфины ближе к коровам, чем к форели [б].

Возможно, для программиста биология представляется зрелой, вполне сформировавшейся наукой с определенными критериями классификации организмов. Но это не так. Биолог Мэй сказал: "На сегодняшний день мы даже не знаем порядок числа видов растений и животных, населяющих нашу планету: классифицировано менее, чем 2 млн. видов, в то время как возможное число видов оценивается от 5 до 50 млн." [7]. Более того, различные критерии классификации одних и тех же животных приводят к разным результатам. Мартин утверждает, что "все зависит от того, что вы хотите получить. Если вы хотите, чтобы классификация говорила о кровном родстве видов, вы получите один ответ, если вы желаете отразить уровень приспособления, ответ будет другой" [8]. Можно заключить, что даже в строгих научных дисциплинах методы и критерии классификации сильно зависят от цели классификации.

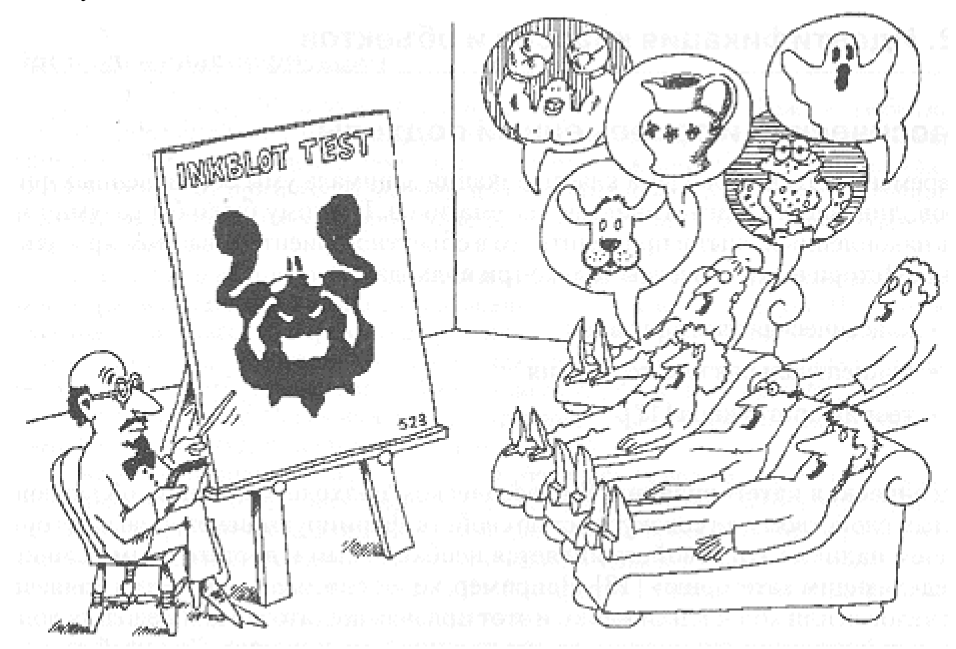
Аналогичная ситуация сложилась и в химии [9]. В древние времена считалось, что все вещества суть комбинации земли, воздуха, огня и воды. В настоящее время такая классификация не может считаться сколько-нибудь удовлетворительной. В середине XVII в. Роберт Бойль предложил элементы как примитивные химические абстракции, из которых составляются более сложные вещества. Век спустя, в 1789 г., Лавуазье опубликовал первый список, содержащий 23 элемента, хотя впоследствии было открыто, что некоторые из них таковыми не являются. Но открытие новых элементов продолжалось, список увеличивался. Наконец, в 1869 г. Менделеев предложил периодический закон, который давал точные критерии для классификации известных элементов и даже мог предсказывать свойства еще не открытых элементов. Но даже периодический закон не был концом истории о классификации элементов. В начале XX в. были открыты элементы с одинаковыми химическими свойствами, но с разными атомными весами - изотопы.

Вывод прост. Как утверждал Декарт: "Открытие порядка - нелегкая задача, но если он найден, понять его совсем не трудно" [10]. Лучшие программистские решения выглядят просто, но, как показывает опыт, добиться простой архитектуры очень трудно.

Итеративная суть классификации. Все эти сведения мы привели здесь не для того, чтобы оправдать "долгострой" в программном обеспечении, хотя на самом деле многим менеджерам и пользователям кажется, что необходимы века, чтобы закончить начатую работу. Мы просто хотели подчеркнуть, что разумная классификация - работа интеллектуальная и лучший способ ее ведения - последовательный, итеративный процесс. Это становится очевидным при анализе разработки таких программных продуктов, как графический интерфейс, стандарты баз данных и языки программирования четвертого поколения. Шоу утверждает, что в разработке программного обеспечения "развитие какой-либо абстракции часто следует общей схеме. В начале проблема решается *ad hoc*, то есть как-нибудь, для каждого частного случая. По мере накопления опыта некоторые решения оказываются более удачными, чем другие, и возникает род фольклора, переходящего от человека к человеку. Удачные решения изучаются более систематически, они программируются и анализируются. Это позволяет развить модели,

осуществить их автоматическую реализацию, и разработать теорию, обобщающую найденное решение. Это в свою очередь поднимает практику на более высокий уровень и позволяет взяться за еще более сложную задачу, к которой, в свою очередь, мы подходим *ad hoc*, тем самым начиная новый виток спирали" [11].

Итеративный подход к классификации накладывает соответствующий отпечаток и на процедуру конструирования иерархии классов и объектов при разработке сложного программного обеспечения. На практике обычно за основу берется какая-то определенная структура классов, которую постепенно совершенствуют.



Разные наблюдатели классифицируют один и тот же объект по-разному

И только на поздней стадии разработки, когда уже получен некоторый опыт использования такой структуры, мы можем критически оценить качество получившейся классификации. Основываясь на полученном опыте, мы можем создать новый подкласс из уже существующих (вывод), или разделить большой класс на много маленьких (факторизация), или, наконец, слить несколько существующих в один (композиция). Возможно, в процессе разработки будут найдены новые общие свойства, ранее не замеченные, и мы сможем определить новые классы (абстракция) [12].

Почему же классификация так сложна? Мы объясняем это двумя причинами. Во-первых, отсутствием "совершенной" классификации, хотя, естественно, одни классификации лучше других. Кумбс, Раффья и Трал утверждают, что "существует столько способов деления мира на объектные системы, сколько ученых принимается за эту задачу" [13]. Любая классификация зависит от точки зрения субъекта. Флуд и Кэрсон приводят пример: "Соединенное Королевство... экономисты могут рассматривать как экономический институт, социологи - как общество, защитники окружающей среды - как гибнущий уголок природы, американские туристы - как достопримечательность, советские руководители - как военную угрозу, наконец, наиболее романтичные из нас, британцев - как зеленые луга родины" [14]. Во-вторых, разумная классификация требует изрядной доли творческого озарения. Бертвистл, Даль, Мюрхауг и Нюгард заключают, что "иногда ответ очевиден, иногда он - дело вкуса, а бывает, что все зависит от умения заметить глазное" [15]. Все это напоминает загадку: "Почему лазерный луч похож на золотую рыбку?.. Потому, что ни тот, ни другой не умеют свистеть" [16]. Надо быть очень творческим мыслителем, чтобы найти общее в настолько несвязанных предметах.

4.2. Идентификация классов и объектов

Классический и современный подходы

Со времен Платона проблема классификации занимала умы бесчисленных философов, лингвистов, когнитивистов, математиков. Поэтому было бы разумно изучить накопленный опыт и применить его в объектно-ориентированном проектировании. Исторически известны только три подхода:

- классическая категоризация
- концептуальная кластеризация
- теория прототипов [17].

Классическая категоризация. В классическом подходе "все вещи, обладающие данным свойством или совокупностью свойств, формируют некоторую категорию. Причем наличие этих свойств является необходимым и достаточным условием, определяющим категорию" [18]. Например, холостые люди - это категория: каждый человек или холост, или женат, и этот признак достаточен для решения вопроса, к какой категории принадлежит тот или иной индивидуум. С другой стороны, высокие люди не определяют категории, если, конечно, мы специально не уточним критерий, позволяющий четко отличать высоких людей от невысоких.

Классическая категоризация пришла к нам от Платона и Аристотеля. Последний в своей классификации растений и животных пользовался техникой рассуждений, напоминающей современную детскую игру в 20 вопросов (Это минерал, животное или растение? Это покрыто мехом или перьями? Может ли оно летать? Пахнет ли оно?) [20]. Такой подход нашел последователей, наиболее выдающимися из которых были: Фома Аквинский, Декарт, Локк. По утверждению Фомы Ак-винского: "Мы можем именовать вещи согласно нашим знаниям об их природе, получаемым через познание их свойств и действий" [21].

Принципы классической категоризации отражены в современной теории развития ребенка. Пьяже утверждает, что после первого года жизни ребенок осознает существование объектов и затем начинает приобретать навыки их классификации, вначале пользуясь базовыми категориями, такими, как собаки, кошки и игрушки [22]. Позднее ребенок осознает, с одной стороны более общие (животные), а с другой стороны, более частные категории (колли, доги, овчарки) [23].

Таким образом, классический подход в качестве критерия похожести объектов использует родственность их свойств. В частности, объекты можно разбивать на непересекающиеся множества в зависимости от наличия или отсутствия некоторого признака. Мински предположил, что "лучшими являются такие наборы свойств, элементы которых мало взаимодействуют между собой. Этим объясняется всеобщая любовь к таким критериям как размер, цвет, форма и материал. Так как эти критерии не пересекаются, про какой-нибудь предмет можно утверждать, что он большой, серый, круглый и деревянный" [24]. Вообще говоря, свойства не обязательно должны быть измеряемыми, в качестве их можно использовать наблюдаемое поведение. То обстоятельство, что птицы летают, а рыбы нет, позволяет отличить орла от форели.

Проблема классификации

На рис. 4-1 показаны 10 поездов, обозначенных буквами от A до J Каждое изображение состоит из паровоза и нескольких вагонов. Прежде чем продолжать чтение, попытайтесь за 10 минут определить несколько групп

изображений, составленных по какому-то логическому признаку. Например, изображения можно разбить на три группы: в одной группе поезда имеют черные колеса, в другой группе - белые, а в третьей - и белые, и черные.

Этот пример взят из работы Степпа и Михальски о концептуальном объединении [19]. Очевидно, "правильного" разбиения на группы не существует. Наши изображения были классифицированы 93 различными способами. Наиболее распространенный способ классификаций по длине состава: были выделены три группы: составы с двумя, тремя и четырьмя вагонами. Второй по популярности вид классификации - по цвету колес поезда. Сорок из девяносто трех видов классификации были уникальными (то есть вид содержал только один экземпляр).

Экспериментируя с этим рисунком, мы убедились в правоте Степпа и Михальски. Большинство опрошенных нами предлагали один из двух наиболее популярных видов классификации (по длине состава и цвету колес поезда). Один опрошенный предложил следующее: в одной группе составы помечены буквами, нарисованными с помощью только прямых линий (A, E, F, H и I), в другой - буквами с кривыми линиями. Вот уж, действительно, пример нетривиального мышления.

Если вы уже справились с заданием, давайте изменим условия. Представим, что круги обозначают груз с токсичными веществами, прямоугольники - лесоматериалы, все остальные знаки обозначают пассажиров. Попытайтесь теперь классифицировать изображения и заметьте, как дополнительная информация влияет на вашу точку зрения.

В наших опытах большинство опрошенных классифицировало поезда по тому, содержит состав токсичный груз или нет. Мы заключили, что новые сведения о реальной ситуации облегчают и улучшают классификацию.

Какие конкретно свойства надо принимать во внимание? Это зависит от обстановки. Например, цвет автомобиля надо зафиксировать в задаче учета продукции автомобилестроительного завода, но он не интересен программе, управляющей уличным светофором. Вот почему мы говорим, что нет абсолютного критерия классификации, одна и та же структура классов может подходить для одной задачи и не годиться для другой. Джеймс пишет: "Нельзя утверждать, что некоторая схема классификации лучше других отражает структуру и порядок вещей в природе. Природе безразличны наши попытки в ней разобраться. Некоторые классификации действительно важнее других, но только в связи с нашими интересами, а не потому, что они вернее или полнее отражают реальность" [25].

Современное западное мышление по большей части насквозь пропитано классической категоризацией, однако, как показывает пример с высокими и низкими людьми, этот подход не всегда работает. Косок отмечает, что "естественные категории не четко отграничены друг от друга. Большинство птиц летает, но не все. Стул может быть деревянным, металлическим или пластмассовым, а количество ног у него целиком зависит от прихоти конструктора. Практически невозможно перечислить определяющие свойства естественной категории, так, чтобы не было исключений" [26]. Это, действительно, коренные пороки классической категоризации, которые и попытались исправить в современных подходах. Ими мы сейчас займемся.

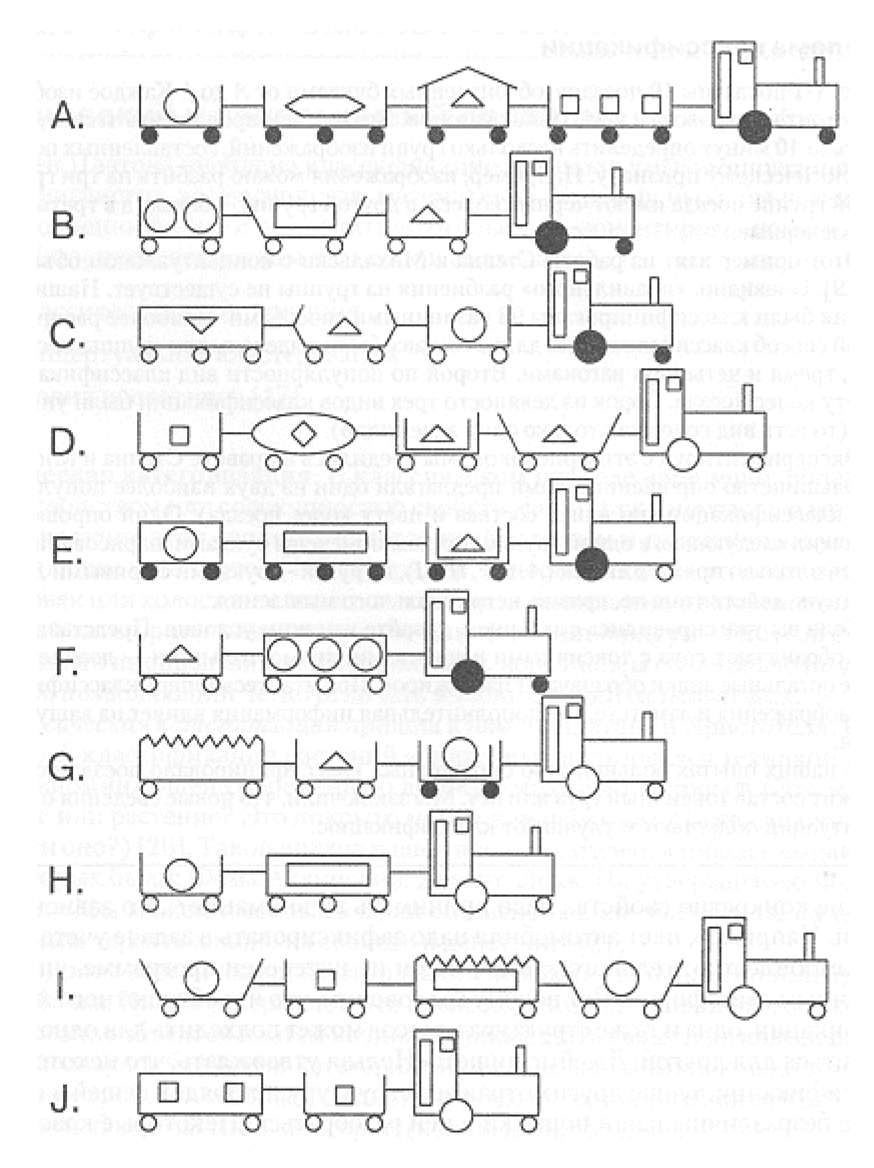


Рис. 4-1. Проблема классификации

Концептуальная кластеризация. Это более современный вариант классического подхода. Он возник из попыток формального представления знаний. Степп и Михальски пишут: "При таком подходе сначала формируются концептуальные описания классов (кластеров объектов), а затем мы классифицируем сущности в соответствии с этими описаниями" [27]. Например, возьмем понятие "любовная песня". Это именно понятие, а не признак или свойство, поскольку степень любов-ности песни едва ли можно измерить. Но если можно утверждать, что песня скорее про любовь, чем про что-то другое, то мы помещаем ее в эту категорию.

Концептуальную кластеризацию можно связать с теорией нечетких (многозначных) множеств, в которой объект может принадлежать к нескольким категориям одновременно с разной степенью точности. Концептуальная кластеризация делает в классификации абсолютные суждения, основываясь на наилучшем согласии.

Теория прототипов. Классическая категоризация и концептуальная кластеризация - достаточно выразительные методы, вполне пригодные для проектирования сложных программных систем. Но все же есть ситуации, в которых эти методы не работают. Рассмотрим более современный метод классификации, теорию прототипов, предпосылки которой можно найти в книге по психологии восприятия Рош и ее коллег[28].

Существуют некоторые абстракции, которые не имеют ни четких свойств, ни четкого определения. Лакофф объясняет эту проблему так: "По утверждению Вит-тгенстейна (Wittgenstein), существуют категории (например, игры), которые не соответствуют классически образцам, так как нет признаков, свойственных всем играм... По этой причине их можно объединить так называемой семейной схожестью... Виттгенстейн утверждает, что у категории игр нет четкой границы. Категорию можно расширить и включить новые виды игр при условии, что они напоминают уже известные игры" [29]. Вот почему этот подход называется теорией прототипов: класс определяется одним объектом-прототипом, и новый объект можно отнести к классу при условии, что он наделен существенным сходством с прототипом.

Лаков и Джонсон применяют классификацию на основе прототипов к упомянутой выше проблеме стульев. Они замечают, что "мы считаем мягкий пуф, парикмахерское кресло и складной стул стульями не потому, что они удовлетворяют некоторому фиксированному набору признаков прототипа, но потому, что они имеют достаточное фамильное сходство с прототипом... Не требуется никакого общего набора свойств прототипа, которое годилось бы и для пуфика и для парикмахерского кресла, но они оба - стулья, так как каждый из них в отдельности похож на прототипный стул, пусть даже каждый по-своему. Свойства, определяемые при взаимодействии с объектом (свойства взаимодействия), являются главными при определении семейного сходства" [30].

Понятие свойств взаимодействия - центральное для теории прототипов. В концептуальной кластеризации мы группируем в соответствии с различными концепциями. В теории прототипов классификация объектов производится по степени их сходства с конкретным прототипом.

Применение классических и новых теорий. Разработчику, озабоченному постоянно меняющимися требованиями к системе и вечно сражающемуся с напряженным планом при ограниченных ресурсах, предмет нашего обсуждения может показаться далеким от реальности. В действительности, три рассмотренных подхода к классификации имеют непосредственное отношение к объектно-ориентированному проектированию.

На практике мы идентифицируем классы и объекты сначала по свойствам, важным в данной ситуации, то есть стараемся выделить и отобрать структуры и типы поведения с помощью словаря предметной области. "Потенциально возможных абстракций, как правило, очень много" [31]. Если таким путем не удалось построить удобоваримой структуры классов, мы пробуем концептуальный подход. В этом случае в центре внимания уделяется поведение объектов, когда они взаимодействуют

друг с другом. Наконец, мы пробуем выделить прототипы и ассоциировать с ними объекты.

Эти три способа классификации составляют теоретическую основу объектно-ориентированного подхода к анализу, предлагающего много практических советов и правил, которые можно применить для идентификации классов и объектов при проектировании сложной программной системы.

Объектно-ориентированный анализ

Границы между стадиями анализа и проектирования размыты, но решаемые ими задачи определяются достаточно четко. В процессе анализа мы моделируем проблему, *обнаруживая* классы и объекты, которые составляют словарь проблемной области. При объектно-ориентированном

проектировании мы *изобретаем* абстракции и механизмы, обеспечивающие поведение, требуемое моделью.[22]

Теперь мы рассмотрим несколько проверенных практикой подходов к анализу объектно-ориентированных систем.

Классические подходы. Разные ученые находят различные источники классов и объектов, согласующихся с требованиями предметной области. Мы называем эти подходы классическими, поскольку они опираются на классическую категоризацию.

Например, Шлаер и Меллор предлагают следующих кандидатов в классы и объекты [32]:

- Осязаемые предметы Автомобили, телеметрические данные, датчики давления
- Роли Мать, учитель, политик
- События Посадка, прерывание, запрос
- Взаимодействие Заем, встреча, пересечение

Что-то в этом роде предлагает Росс, исходя из перспектив моделирования баз данных [33]:

- Люди Человеческие существа, выполняющие некоторые функции
- Места Области, связанные с людьми или предметами
- Предметы Осязаемый материальный объект или группа объектов
- Организации Формально организованная совокупность людей, ресурсов, оборудования, которая имеет определенную цель и существование которой в целом не зависит от индивидуумов
- Концепции Принципы и идеи, сами по себе неосязаемые, но предназначенные для организации деятельности и/или общения, или же для наблюдения за ними
- События Нечто случающееся с чем-то в заданное время или последовательно

Коад и Иордан предложили свой список [34]:

- Структуры Отношения "целое-часть" и "общее-частное"
- Другие системы Внешние системы, с которыми взаимодействует приложение
- Устройства Устройства, с которыми взаимодействует приложение
- События Происшествия, которые должны быть запомнены
- Разыгрываемые роли Роли, которые исполняют пользователи, работающие с приложением

---

[22] Обозначения и процессы, описанные в этой книге, в равной степени относятся к фазам и анализа и проектирования (в традиционном понимании), как мы увидим в главе 6. Именно по этой причине мы сменили во втором издании название книги на "Объектно-ориентированный анализ и проектирование".

- Места Здания, офисы и другие места, существенные для работы приложения
- Организационные Группы, к которым принадлежат пользователи. единицы

На более высоком уровне абстракции Коад вводит понятие предметной области, которая в сущности является логически связанной группой классов, относящейся к высокоуровневым функциям системы.

Анализ поведения. В то время как классические подходы концентрируют внимание на осязаемых элементах предметной области, другая школа мысли объектно-ориентированного анализа сосредотачивается на динамическом поведении как на первоисточнике объектов и классов.[23] Это напоминает концептуальную кластеризацию, рассмотренную выше: мы формируем классы, основываясь на группах объектов, демонстрирующих сходное поведение.

Вирфс-Брок предлагает понятие ответственности объекта, под которыми следует понимать "его знания и умения. Ответственность - это способ выразить цель объекта и его место в системе. Ответственность объекта есть совокупность всех услуг, которые он может предоставлять по всем его контрактам" [36]. То есть, мы объединяем вместе те объекты, которые имеют сходные ответственности и строим иерархию классов, в которой каждый подкласс, выполняя обязательства суперкласса, привносит свои дополнительные услуги.

Рубин и Гольдберг предлагают идентифицировать классы и объекты, анализируя функционирование системы: "Наш подход основан на изучении поведения системы. Мы сопоставляем формы поведения с частями системы и пытаемся понять, какая часть инициирует поведение и какие части в нем участвуют... Инициаторы и участники, играющие существенные роли, опознаются как объекты и делаются ответственными за эти роли" [37].

Идеи Рубина тесно связаны с предложенным в 1979 году Альбрехтом подходом с точки зрения функций. По его определению, функция "определяется как отдельное бизнес-действие конечного пользователя" [38], то есть: ввод/вывод, запрос, файл или интерфейс. Очевидно, что эта концепция происходит из области информационных систем. Однако, она может быть применена к любой автоматизированной системе. По существу, функция - это любое достоверно видимое извне и имеющее отношение к делу поведение системы.

Анализ предметной области. До сих пор мы неявно имели в виду единственное разрабатываемое нами приложение. Но иногда в поисках полезных и уже доказавших свою работоспособность идей полезно обратиться сразу ко всем приложениям в рамках данной предметной области, как, например, ведение историй болезни пациентов, торговля ценными бумагами, разработка компиляторов или системы управления ракетами. Если вы находитесь в середине разработки и застряли, анализ какой-нибудь узкой предметной области может помочь, указав вам на ключевые абстракции, оказавшиеся полезными в сходных системах. Анализ предметной области работает очень хорошо, исключая разве что лишь очень специальные ситуации, так как уникальные программные системы встречаются крайне редко.

---

[23] Шлаер и Меллор дополнили свою более раннюю работу, обратив внимание также и на поведение. В частности, они изучали жизненный цикл объекта как средство понимания границ.

Идею анализа предметной области впервые предложил Нейборс. Мы определим такой анализ как "попытку выделить те объекты, операции и связи, которые эксперты данной области считают наиболее важными" [39]. Мур и Байлин определяют следующие этапы в анализе области:

- "Построение скелетной модели предметной области при консультациях с экспертами в этой области.
- Изучение существующих в данной области систем и представление результатов в стандартном виде.
- Определение сходства и различий между системами при участии экспертов.
- Уточнение общей модели для приспособления к нуждам конкретной системы" [40].

Анализ области можно вести относительно аналогичных приложений (вертикально) или относительно аналогичных частей одного и того же приложения (горизонтально). Например, начиная проектировать систему учета пациентов, имеет смысл рассмотреть уже имеющиеся подобные системы, чтобы понять, какие ключевые абстракции и механизмы, использованные в них, будут вам полезны, а какие нет. Аналогично система бухгалтерского учета должна представлять различные виды отчетов. Если считать отчеты некой предметной областью, ее анализ может привести разработчика к пониманию ключевых абстракций и механизмов, которые обслуживают все виды отчетов. Полученные таким образом классы и объекты представляют собой множество ключевых абстракций и механизмов, отобранных с учетом цели исходной задачи: создания системы отчетов. Поэтому окончательный проект будет проще.

Определим теперь, кто такой эксперт? В роли эксперта часто выступает просто пользователь системы, например, инженер или диспетчер. Он не обязательно должен быть программистом, но должен быть близко знаком с исследуемой проблемой и разговаривать на языке этой проблемы.

Менеджеры проектов заинтересованы в непосредственном сотрудничестве пользователей и разработчиков системы. Но для очень сложных систем прикладной анализ является формальным процессом, для которого требуется большое число экспертов и разработчиков на длительный период времени. На практике такой формальный анализ требуется редко. Обычно для начального уяснения проблемы достаточно короткой встречи экспертов и разработчиков. Удивительно, как мало информации требуется для продуктивной работы разработчика. Однако мы считаем чрезвычайно полезными такие встречи в течение всей разработки. Анализ прикладной области лучше всего вести шаг за шагом - немного поанализировать, потом немного попроектировать и т. д.

Анализ вариантов. По отдельности классический подход, поведенческий подход и изучение предметной области, рассмотренные выше, сильно зависят от индивидуальных способностей и опыта аналитика. Для большинства реальных проектов одновременное применение всех трех подходов неприемлемо, так как процесс анализа становится недетерминированным и непредсказуемым.

Анализ вариантов - это подход, который можно успешно сочетать с первыми тремя, делая их применение более упорядоченным. Впервые его формализовал Джекобсон, определивший вариант применения, как "частный пример или образец использования, сценарий, начинающийся с того, что пользователь системы инициирует операцию или последовательность взаимосвязанных событий" [41].

Коротко говоря, этот вид анализа можно начинать вместе с анализом требований. В этот момент пользователи, эксперты и разработчики

перечисляют сценарии, наиболее существенные для работы системы (пока не углубляясь в детали). Затем они тщательно прорабатывают сценарии, раскладывая их по кадрам, как делают телевизионщики и кинематографисты [42]. При этом они устанавливают, какие объекты участвуют в сценарии, каковы обязанности каждого объекта и как они взаимодействуют в терминах операций. Тем самым группа разработчиков вынуждена четко распределить области влияния абстракций. Далее набор сценариев расширяется, чтобы учесть исключительные ситуации и вторичное поведение (Гольд-стейн и Алджер называют это периферийными аспектами [43]). В результате появляются новые или уточняются существующие абстракции. Позже, в главе 6, мы покажем, как сценарии используются для тестирования.

CRC-карточки. CRC обозначает Class-Responsibilities-Collaborators (Класс/Ответственности/Участники). Это простой и замечательно эффективный способ анализа сценариев. Карты CRC впервые предложили Бек и Каннингхэм для обучения объектно-ориентированному программированию, но такие карточки оказались отличным инструментом для мозговых атак и общения разработчиков между собой.

Собственно, это обычные библиографические карточки 3x5 дюйма (если позволяет бюджет вашего проекта, купите 5x7; очень хорошо, если карточки будут линованными, а разноцветные - просто мечта). На карточках вы пишите (обязательно карандашом) сверху - название класса, снизу в левой половине - за что он отвечает, а в правой половине - с кем он сотрудничает. Проходя по сценарию, заводите по карточке на каждый обнаруженный класс и дописывайте в нее новые пункты. При этом каждый раз обдумывайте, что из этого получается, и "выделяйте излишек ответственности" в новый класс или, что случается чаще всего, перенесите ответственности с одного большого класса на несколько более детальных классов, или, возможно, передайте часть обязанностей другому классу.

Карточки можно раскладывать так, чтобы представить формы сотрудничества объектов. С точки зрения динамики сценария, их расположение может показать поток сообщений между объектами, с точки зрения статики они представляют иерархии классов.

Неформальное описание. Радикальная альтернатива классическому анализу была предложена в чрезвычайно простом методе Аббота. Согласно этому методу надо описать задачу или ее часть на простом английском языке, а потом подчеркнуть существительные и глаголы [45]. Существительные - кандидаты на роль классов, а глаголы могут стать именами операций. Метод можно автоматизировать, и такая система была построена в Токийском технологическом институте и в Fujitsu [46].

Подход Аббота полезен, так как он прост и заставляет разработчика заниматься словарем предметной области. Однако он весьма приблизителен и непригоден для сколько-нибудь сложных проблем. Человеческий язык - ужасно неточное средство выражения, потому список объектов и операций зависит от умения разработчика записывать свои мысли. Тем более, что для многих существительных можно найти соответствующую глагольную форму и наоборот.

Структурный анализ. Вторая альтернатива классической технике объектно-ориентированного анализа использует структурный анализ как основу для объектно-ориентированного проектирования. Такой подход привлекателен потому, что много аналитиков применяют этот подход и имеется большое число программных CASE-средств, поддерживающих автоматизацию этих методов. Нам лично не нравится использовать структурный анализ как основу для объектно-ориентированного

проектирования, но для некоторых организаций такой прагматический подход не имеет альтернативы.

После проведения структурного анализа мы уже имеем модель системы, описанную диаграммами потоков данных и другими продуктами структурного анализа. Эти диаграммы дают нам формальную модель проблемы. Исходя из модели, мы можем приступить к определению осмысленных классов и объектов тремя различными способами.

МакМенамин и Палмер предлагают сначала приступить к формированию словаря данных и затем к анализу контекстных диаграмм модели. Они говорят:

"рассматривая список основных структур данных, следует подумать, о чем они говорят или что описывают. Например, если они прилагательные, то какие существительные они описывают? Ответы на такие вопросы могут пополнить ваш список объектов" [47]. Эти кандидаты в объекты происходят из окружающей среды, из существенных входных и выходных данных, а также продуктов, услуг и других ресурсов, которыми она управляет.

Следующие два способа основаны на анализе отдельных диаграмм потоков данных. Если взять какую-нибудь диаграмму потоков (в терминологии Барда и Меллора [48]), то кандидаты в объекты это:

- внешние сущности
- хранилища данных
- хранилища управляющих сущностей
- управляющие преобразования.

Кандидаты в классы:

- потоки данных
- потоки управления.

Остается преобразование данных, которое мы можем рассматривать как операции над существующими объектами или как поведение некоторого объекта, который мы создали специально для выполнения нужного преобразования.

Зайдевиц и Старк предлагают еще один метод, который они называют анализом абстракций. Метод базируется на идентификации основных сущностей, которые по своей природе аналогичны основным преобразованиям в структурном проектировании. Как они говорят, "в структурном анализе входные и выходные данные изучаются до тех пор, пока не достигнут высшего уровня абстракции. Процесс преобразования входных данных в выходные есть основное преобразование. В абстрактном анализе разработчик делает то же самое, а также изучает основное преобразование для того, чтобы определить, какие процессы и состояния представляют наилучшую абстрактную модель системы" [49]. После определения основной сущности в диаграмме потоков данных аналитик приступает к изучению всей инфраструктуры, прослеживая входящие и исходящие потоки данных из центра, группируя процессы и состояния, встречающиеся по пути. Для практического использования авторы нашли анализ абстракций слишком сложным и в качестве альтернативы предлагают объектно-ориентированный анализ [50].

Необходимо отметить, что принципы структурного проектирования, которое, естественно, следует за структурным анализом, полностью ортогональны принципам объектно-ориентированного проектирования. Наш опыт показывает, что использование структурного анализа в процессе объектно-ориентированного проектирования часто приводит к полному провалу, если разработчик не способен сопротивляться желанию свалиться в структурную пропасть. Другая очень серьезная опасность заключается в том, что многие аналитики любят рисовать диаграммы потоков данных, которые

представляют собой скорее описание проекта, чем модель существа системы. Очень трудно построить объектно-ориентированную систему, если модель столь очевидно ориентирована на алгоритмическую декомпозицию. Поэтому мы предпочитаем объектно-ориентированный анализ и анализ проблемной области как подготовительный этап для объектно-ориентированного проектирования. При этом уменьшается риск замусорить проект элементами алгоритмического анализа.

Если же по каким-либо уважительным причинам[24] приходится взять за основу структурный анализ, прекратите строить диаграммы, как только они начинают смахивать на проект программы, а не на модель предметной области. Помните, что материалы проектирования, такие, как диаграммы потоков данных, это не конечный продукт, а инструмент разработчиков. Обычно строятся диаграммы, а затем разрабатываются механизмы, обеспечивающие необходимое поведение системы, то есть сам акт проектирования видоизменяет начальную модель. Поддержание соответствия модели и развивающегося проекта - дело трудное, и, честно говоря, бесполезное. Имеет смысл сохранять только продукт структурного анализа высокого уровня абстракции. Он отражает существенные черты и достаточно независим от проекта системы.

4.3. Ключевые абстракции и механизмы

Ключевые абстракции

Поиск и выбор ключевых абстракций. *Ключевая абстракция* - это класс или объект, который входит в словарь проблемной области. Самая главная ценность ключевых абстракций заключена в том, что они определяют границы нашей проблемы: выделяют то, что входит в нашу систему и поэтому важно для нас, и устраняют лишнее. Задача выделения таких абстракций специфична для проблемной области. Как утверждает Голдберг, "правильный выбор объектов зависит от назначения приложения и степени детальности обрабатываемой информации" [51].

Как мы уже отмечали, определение ключевых абстракций включает в себя два процесса: открытие и изобретение. Мы открываем абстракции, слушая специалистов по предметной области: если эксперт про нее говорит, то эта абстракция обычно действительно важна [52]. Изобретая, мы создаем новые классы и объекты, не обязательно являющиеся частью предметной области, но полезные при проектировании или реализации системы. Например, пользователь банкомата говорит "счет, снять, положить"; эти термины - часть словаря предметной области. Разработчик системы использует их, но добавляет свои, такие, как база данных, диспетчер экрана, список, очередь и так далее. Эти ключевые абстракции созданы уже не предметной областью, а проектированием.

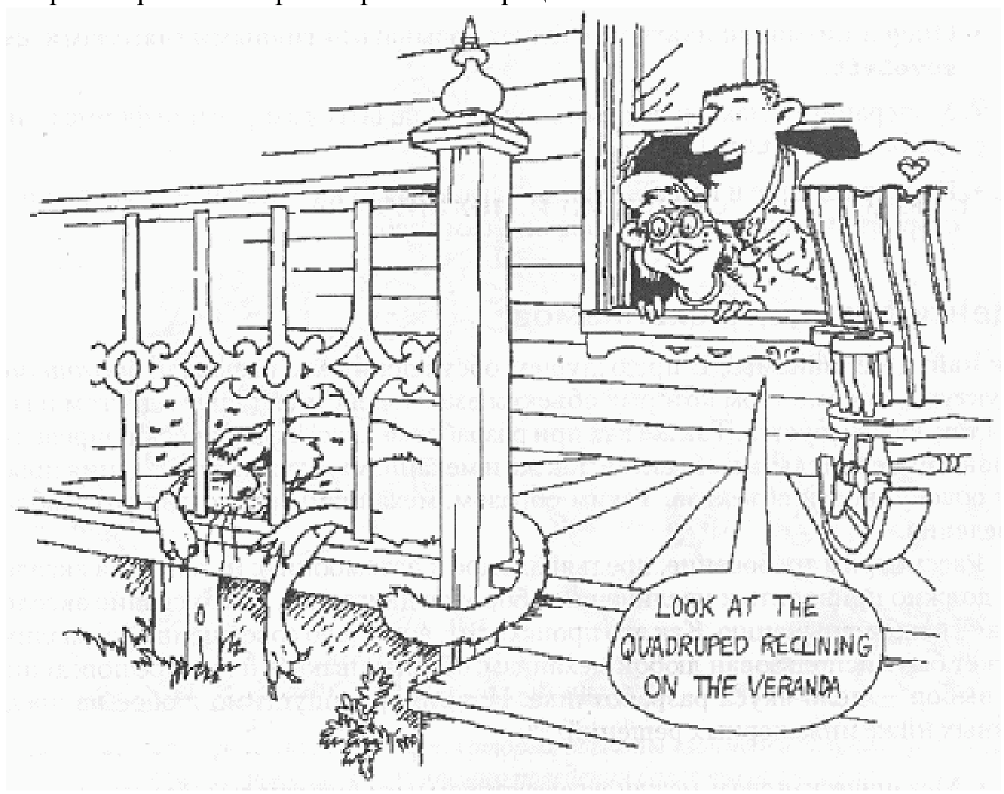
Наиболее мощный способ выделения ключевых абстракций - сводить задачу к уже известным классам и объектам. Как будет показано ниже в главе 6, при отсутствии таких повторно используемых абстракций мы рекомендуем пользоваться сценариями, чтобы вести процесс идентификации классов и объектов.

Уточнение ключевых абстракций. Определив кандидатов на роли ключевых абстракций, мы должны оценить их по критериям, описанным в предыдущих главах. По словам Страуструпа "программист должен задаваться вопросами: Как создаются объекты класса? Как можно копировать и/или уничтожать объекты данного класса? Какие операции могут быть выполнены над этим объектом? Если ответы на эти вопросы туманны, то, возможно,

[24] Политические и исторические причины в качестве уважительных не принимаются.

общая концепция не ясна и лучше сесть и подумать еще раз, чем бросаться программировать" [53].

Определив новые абстракции, мы должны найти их место в контексте уже существующих классов и объектов. Не стоит пытаться делать это строго сверху вниз или снизу вверх. Халберт и О'Брайен утверждают, что "нет особой необходимости строить иерархию классов, начиная с самого верхнего класса, и потом дополнять ее подклассами. Чаще вы создаете несколько независимых иерархий, осознаете их общие черты и выделяете один или несколько суперклассов. Требуется несколько проходов вверх и вниз по иерархии, чтобы создать программный проект" [54]. Это не карт-бланш на хакерство, а всего лишь наблюдение, основанное на опыте и подтверждающее тот факт, что объектно-ориентированное проектирование - процесс



Классы и объекты должны быть на надлежащем уровне абстракции: не слишком высоко и не слишком низко

последовательных приближений. Сходное наблюдение делает Страуструп: "Наиболее частые реорганизации в иерархии классов - это сведение совпадающих частей двух классов в один и разделение класса на два новых" [55].

Трудно сразу расположить классы и объекты на правильных уровнях абстракции. Иногда, найдя важный класс, мы можем передвинуть его вверх в иерархии классов, тем самым увеличивая степень повторности использования кода. Это называется продвижением класса [56]. Аналогично, можем прийти к выводу, что класс слишком обобщен, и это затрудняет наследование: происходит семантический разрыв или конфликт зернистости [57]. В обоих случаях мы пытаемся выявить зацепление или недостаточную связность абстракций и смягчить конфликт.

Программисты часто легкомысленно относятся к правильному наименованию классов и объектов, но на самом деле очень важно отразить в обозначении классов и объектов сущность описываемых ими предметов. Программы необходимо писать тщательно, как художественную литературу, дума я и о читателях, и о компьютере [58]. При идентификации одного только объекта вам нужно придумать имена: для него, для его класса и для модуля, в

котором класс объявлен. Умножьте на тысячу объектов и сотни классов, и вы поймете, как остра проблема.

Мы предлагаем следующие правила:

- Объекты следует называть существительными: **theSensor** или **shape**.
- Классы следует называть обобщенными существительными: **Sensors**, **Shapes**.
- Операции-модификаторы следует называть активными глаголами: **Draw**, **moveLeft**.
- У операций-селекторов в имя должен включаться запрос или форма глагола "to be": **extentOf**, **isOpen**.
- Подчеркивание и использование заглавных букв - на ваше усмотрение, постарайтесь лишь не противоречить сами себе.

Идентификация механизмов

Как найти механизмы? В предыдущем обсуждении мы называли механизмами структуры, посредством которых объекты взаимодействуют друг с другом и ведут себя так, как требуется. Так же как при разработке класса фактически определяется поведение отдельных объектов, так же и механизмы служат для задания поведения совокупности объектов. Таким образом, механизмы представляют шаблоны поведения.

Рассмотрим требование, предъявляемое к автомобилю: нажатие на акселератор должно приводить к увеличению оборотов двигателя, а отпускание акселератора - к их уменьшению. Как это происходит, водителю совершенно безразлично. Может быть использован любой механизм, обеспечивающий нужное поведение, и его выбор - дело вкуса разработчика. Например, допустимо любое из предложенных ниже инженерных решений:

- Механическая связь между акселератором и карбюратором (обычное решение).
- Под педалью ставится датчик давления, который соединяется с компьютером, управляющим карбюратором (механизм управления по проводам).
- Карбюратора нет. Бак с горючим находится на крыше автомобиля и топливо свободно течет в двигатель. Поток топлива регулируется зажимом на трубке. Нажатие на педаль акселератора ослабляет зажим (очень дешево).

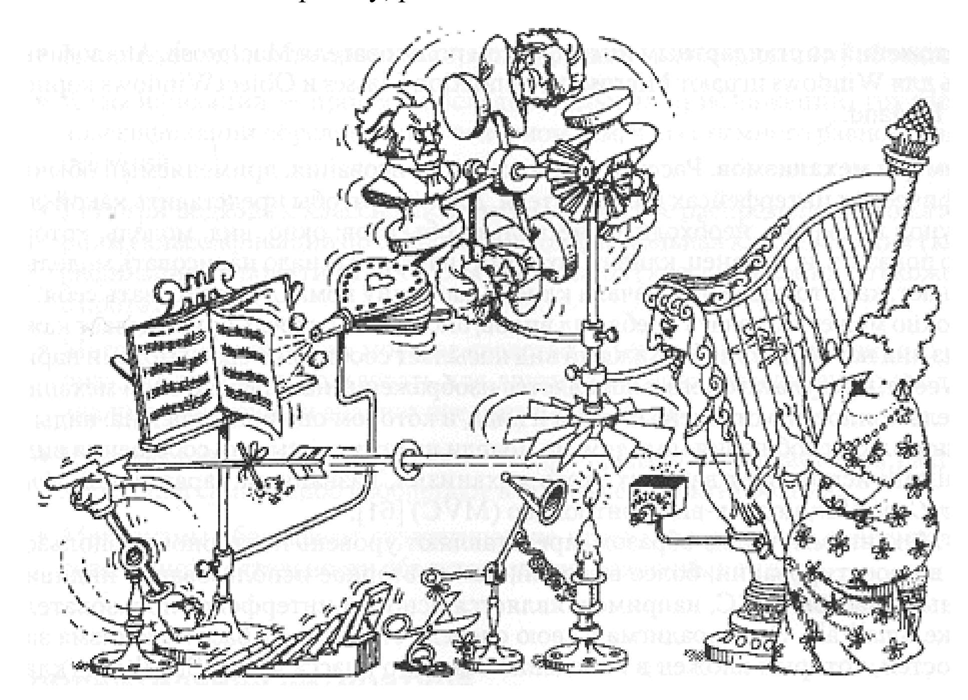
Какую именно реализацию выберет разработчик, зависит от таких параметров, как стоимость, надежность, технологичность и т. д.

Подобно тому, как было бы недопустимой невежливостью со стороны клиента нарушать правила пользования сервером, также и выход за пределы правил и ограничений поведения, заданных механизмом, социально неприемлем. Водитель был бы удивлен, если бы, нажав на педаль акселератора, он увидел зажегшиеся фары.

Ключевые абстракции определяют словарь проблемной области, механизмы определяют суть проекта. В процессе проектирования разработчик должен придумать не только начинку классов, но и то, как объекты этих классов будут взаимодействовать друг с другом. Но конкретный механизм взаимодействия все равно придется разложить на методы классов. В итоге протокол класса будет отражать поведение его объектов и работу механизмов, в которых они участвуют.

Механизмы, таким образом, представляют собой стратегические решения в проектировании, подобно проектированию структуры классов. С другой стороны, проектирование интерфейса какого-то одного класса - это

скорее тактическое решение. Стратегические решения должны быть выполнены явно, иначе у нас получится неорганизованная толпа объектов, кидающихся выполнять работу, растал-



Механизмы суть средства, с помощью которых объекты взаимодействуют друг с другом для достижения необходимого поведения более высокого уровня

кивая друг друга. В наиболее элегантных, стройных и быстрых программах воплощены тщательно разработанные механизмы.

Механизмы представляют только один из шаблонов, которые мы находим в структурированных системах. Так, на нижнем конце своеобразной биологической пирамиды находятся идиомы. Это обороты, специфические для языков программирования или программистских культур, и отражающие общепринятые способы выражаться.[25] Например, в CLOS не принято использовать подчеркивание в именах функций или переменных, хотя в Ada это дело обычное [59]. Изучая язык, приходится учить его идиомы, которые обычно передаются в форме фольклора. Однако, как отметил Коплейн, идиомы играют важную роль в кодификации шаблонов низкого уровня. Он заметил, что "многие общепрограммистские действия идиоматичны" и поэтому распознание таких идиом позволяет "использовать конструкции C++ для выражения функциональности вне самого этого языка с сохранением иллюзии, что они являются частью языка" [60].

Место на верху пирамиды занимают среды разработки. Среда разработки - это собрание классов, предназначенных для определенной прикладной ситуации. Среда дает готовые классы, механизмы и услуги, которыми можно сразу пользоваться или приспосабливать для своих нужд.

Если идиомы составляют часть программистской культуры, то среды разработки обычно - коммерческий продукт. Например, Apple MacApp и его преемник Bedrock - среды, написанные на C++ и предназначенные для построения приложений со стандартным интерфейсом пользователя Macintosh. Аналогичную роль для Windows играют Microsoft Foundation Classes и ObjectWindows корпорации Borland.

[25] Определяющей характеристикой идиомы является то, что ее игнорирование или нарушение влечет немедленные социальные последствия: вы превращаетесь в йеху или, еще хуже, в чужака, не заслуживающего уважения.

Примеры механизмов. Рассмотрим механизм рисования, применяемый обычно в графических интерфейсах пользователя. Для того, чтобы представить какой-либо рисунок на экране, необходимы несколько объектов: окно, вид, модель, которую надо показать, и, наконец, клиент, который знает, когда надо нарисовать модель, но не знает, как это сделать. Сначала клиент дает окну команду нарисовать себя. Так как окно может включать в себя ряд видов, оно в свою очередь приказывает каждому из них нарисовать себя. Каждый вид посылает сообщение своей модели нарисовать себя, в результате чего и появляется изображение на экране. В этом механизме модель полностью отделена от окна и вида, в котором она представлена: виды могут посылать сообщения моделям, но модели не могут посылать сообщения видам. Smalltalk использует вариант этого механизма, названный парадигмой Model-View-Controller, модель-вид-контроллер (MVC) [61].

Механизмы, таким образом, представляют уровень повторного использования в проектировании, более высокий, чем повторное использование индивидуальных классов. MVC, например, является основой интерфейса пользователя в языке Smalltalk. Эта парадигма в свою очередь строится на базе механизма зависимостей, который вложен в поведение базового класса языка Smalltalk (класса object) и часто используется библиотекой классов языка Smalltalk.

Примеры механизмов можно найти во многих системах. Структуру операционной системы, например, можно описать на высоком уровне абстракции по тем механизмам, которые используются для диспетчеризации программ. Система может быть монолитной (как MS-DOS), иметь ядро (UNIX) или представлять собой иерархию процессов (операционная система THE) [62]. В системах искусственного интеллекта использованы разнообразные механизмы принятия решений. Одним из наиболее распространенных является механизм рабочей области, в которую каждый индивидуальный источник знаний независимо заносит свои сведения. В таком механизме не существует центрального контроля, но любое изменение в рабочей области может явиться толчком для выработки системой нового пути решения поставленной задачи [63]. Коад похожим образом выявил ряд общих механизмов в объектно-ориентированных системах, включая шаблоны временных ассоциаций, протоколирование событий и широковещательную рассылку сообщений [64]. Во всех случаях эти механизмы проявляются не как индивидуальные классы, а как структуры сотрудничающих классов.

На этом завершается наше изучение классификации и понятий, являющихся основой объектно-ориентированного проектирования. Следующие три главы посвящены самому методу, в частности системе обозначений, процессу проектирования и рассмотрению практических примеров.

Выводы

- Идентификация классов и объектов - важнейшая задача объектно-ориентированного проектирования; процесс идентификации состоит из открытия и изобретения.
- Классификация есть проблема группирования (кластеризации) объектов.
- Классификация - процесс последовательных приближений; трудности классификации обусловлены в основном тем, что есть много равноправных решений.
- Есть три подхода к классификации: классическое распределение по категориям (классификация по свойствам), концептуальная

кластеризация (классификация по понятиям) и теория прототипов (классификация по схожести с прототипом).

- Метод сценариев - это мощное средство объектно-ориентированного анализа, его можно использовать для других методов: классического анализа, анализа поведения и анализа предметной области.
- Ключевые абстракции отражают словарь предметной области; их находят либо в ней самой, либо изобретают в процессе проектирования.
- Механизмы обозначают стратегические проектные решения относительно совместной деятельности объектов многих различных типов.

Дополнительная литература

Проблема классификации вечна. В своей работе "Политик" Платон вводит классический подход к классификации, группируя объекты со схожими свойствами. Аристотель в "Категориях" продолжает эту тему и анализирует различие между классами и объектами. Несколькими веками позже Фома Аквинский в "Summa Theologica" и затем Декарт в "Рассуждении о методе" обдумывают философию классификации. Среди современных объективистских философов можно назвать Рэнда (Rand) [I 1979].

Альтернативы объективистскому взгляду на мир обсуждаются Лаковым (Lakoff) [I 1990] и Голдстейном и Алжером (Goldstein and Alger) [C 1992].

Умение классифицировать - важный человеческий навык. Теории приобретения этого навыка в раннем детстве строились первоначально Пьяже (Piaget) и были подытожены Майером (Maier) [A 1969]. Лефрансуа (Lefrancois) [A 1977] дал легко читаемое введение в эти идеи и блестяще изложил процесс формирования у детей концепции объекта.

Когнитивисты изучили проблему классификации во всех деталях. Ньэлл и Саймон (Newell and Simon) [A 1972] дали ни с чем не сравнимый источник материала по человеческим навыкам классификации. Более подробная информация может быть найдена в работах Саймона (Simon) [A 1982], Хофстадтера (Hofstadter) [I 1979], Зиглера и Ричардса (Siegler and Richards) [A 1982] и Стиллинга и др. (Stillings et al.) [A 1987]. Лингвист Лаков (Lakoff) [A 1988] анализировал способы, которыми разные человеческие языки справляются с проблемами классификации и что это говорит о мышлении. Мински (Minsky) [A 1986] подошел к этому вопросу с другой стороны, от теории структуры сознания.

Концептуальную кластеризацию как подход к представлению знания через классификацию в деталях описали Михальски и CTenn(Michalski and Stepp) [A 1983, 1986],Пекхам и Марьянский (Peckham and Maryanski) [] 1988] и Coya (Sowa) [A 1984]. Анализ предметных областей, подход к выделению ключевых абстракций и механизмы изучения словаря предметной области описаны во всеобъемлющем собрании работ Прието-Диа-са и Аранго (Prieto-Diaz and Arango) [A 1991]. Иско (Iscoe) [B 1988] принадлежит несколько важных достижений в этой области. Дополнительная информация может быть

найдена в работах Иско, Броуна и Вета (Iscoe, Browne and Weth) [B 1989], Мура и Бэй-лина (Moore and Bailin) [B 1988] и Аранго (Arango) [B 1989].

Интеллектуальная классификация часто требует нового, нестандартного взгляда на мир, и этому искусству можно научить. Фон Оич (Von Oech) [I 1990] предлагает некоторые пути развития творческих способностей. Коад (Coad) [A 1993] создал настольную игру Object Game, способствующую развитию навыков идентификации классов и объектов.

Хотя эта область пребывает еще в младенческом состоянии, но некоторая многообещающая работа по каталогизации шаблонов уже проведена. В частности, выявлены идиомы, механизмы и среды разработки. Интересные ссылки: Коплиен (Coplien) [G 1992], Коад (Coad) [A 1992], Джонсон (Johnson) [A 1992], Шоу (Shaw) [A 1989,1990, 1991], Вирфс-Брок (Wirfs-Brock) [C 1991]. Работа Александера (Alexander) [I 1979] посвящена применению шаблонов в архитектуре и городском планировании.

Математики пытались развить эмпирические подходы к классификации, доведя их до того, что называется теорией измерения. Стивене (Stevens) [A 1946] и Кумбс, Райфа и Тралл (Coombs, Raiffa and Thrall) [A 1954] провели в этом направлении плодотворную работу.

Классификационное Общество Северной Америки издает журнал с периодичностью два выпуска в год, содержащий множество статей по вопросам классификации.

ЧАСТЬ ВТОРАЯ

Метод

Какое нововведение приведет к успешному проекту, а какое к провалу не вполне предсказуемо. Каждая возможность создать что-то новое, будь то мост, самолет или небоскреб, ставит инженера перед выбором, который может казаться безграничным. Он может заимствовать сколько захочет все лучшее и полезное из тех существующих проектов, которые успешно противостоят разрушающим силам человека и природы, а может, напротив, решить улучшить те аспекты предыдущих проектов, которые сочтет нужным.

Генри Петроски (Henry Petroski)
Проектирование как человеческая деятельность
(To Engineer is Human)

Глава 5

Обозначения

Составление диаграмм - это еще не анализ и не проектирование. Диаграммы позволяют описать поведение системы (для анализа) или показать детали архитектуры (для проектирования). Если вы понаблюдаете за работой инженера (программиста, технолога, химика, архитектора), вы быстро убедитесь, что будущая система формируется в сознании разработчика и только в нем. Когда, спустя некоторое время, система будет понятна в общих чертах, она скорее всего будет представлена на таких высокотехнологичных носителях, как тетрадные листы, салфетки или старые конверты [1].

Однако, хорошо продуманная и выразительная система обозначений очень важна для разработки. Во-первых, общепринятая система позволяет разработчику описать сценарий или сформулировать архитектуру и доходчиво изложить свои решения коллегам. Символ транзистора понятен всем электронщикам мира. Аналогично, над проектом жилого дома, разработанным архитекторами в Нью-Йорке, строителям из Сан-Франциско скорее всего не придется долго ломать голову, решая, как надо расположить двери, окна или электрическую проводку. Во-вторых, как подметил Уайтхед в своей основополагающей работе по математике, "Освобождая мозг от лишней работы, хорошая система обозначений позволяет ему сосредоточиться на задачах более высокого порядка" [2]. В-третьих, четкая система обозначений позволяет автоматизировать большую часть утомительной проверки на полноту и правильность. Как говорится в отчете управления оборонных исследований:

"Разработка программного обеспечения всегда будет кропотливой работой... Хотя наши машины могут выполнять рутинную работу и помогать нам держать нить рассуждений, разработка концепций останется прерогативой человека... Что всегда будет в цене - это искусство построения концептуальной структуры, а заботы об ее описании уйдут в прошлое" [3].

5.1. Элементы обозначений

Необходимость разных точек зрения

Невозможно охватить все тонкие детали сложной программной системы одним взглядом. Как отмечают Клейн и Джингрич: "Необходимо понять как функциональные, так и структурные свойства объектов. Следует уяснить также таксономическую структуру классов объектов, используемые механизмы наследования, индивидуальное поведение объектов и динамическое поведение системы в целом. Эта задача в чем-то аналогична показу футбольного или теннисного матча, когда для вразумительной передачи происходящего действия требуется несколько камер, расположенных в разных углах спортивной площадки. Каждая камера передает свой аспект игры, недоступный другим камерам" [4].

На рис. 5-1 представлены различные типы моделей (описанные в главе 1), которые мы считаем главными в объектно-ориентированном подходе. Через них будут выражаться результаты анализа и проектирования, выполненные в рамках любого проекта. Эти модели в совокупности семантически достаточно богаты и универсальны, чтобы разработчик мог выразить все заслуживающие внимания стратегические и тактические решения, которые он должен принять при анализе системы и формировании ее архитектуры. Кроме того, эти модели достаточно полны, чтобы служить техническим проектом реализации практически на любом объектно-ориентированном языке программирования.

Наша система обозначений богата деталями, но это не означает, что в каждом проекте необходимо использовать все ее аспекты. На практике для описания большей части итогов анализа и проектирования достаточно

некоторого малого подмножества этой системы; один наш коллега называет такие облегченные обозначения "нотацией *Booch Lite*" (сокращенная нотация Буча). В процессе знакомства с нашей системой обозначений мы четко выделим это подмножество. Зачем же тогда беспокоиться о деталях системы обозначений, не вошедших в минимальное подмножество? В основном они нужны, чтобы выражать некоторые важные тактические решения; а некоторые из них предназначены для создания программных инструментов CASE. Мы будем называть их *дополнениями*.

Как отмечает Вайнберг: "В некоторых областях (таких, как архитектура), в процессе проектирования графический план чаще всего представлен в виде общих набросков, а строго детализированные описания, пока не окончена творческая часть становления проекта, используются редко" [5]. Следует помнить, что обозначения - только средство выразить итоги размышлений над архитектурой и поведением системы, а никак не самоцель. Надо пользоваться исключительно теми элементами обозначений, которые необходимы, и не более того. Также, как опасно ставить слишком жесткие требования, нехорошо чересчур подробно описывать решение задачи. Например, на чертеже архитектор может показать расположение выключателя света в комнате, но его точное место не будет определяться, пока заказчик и прораб не произведут осмотр уже сооруженного здания для уточнения

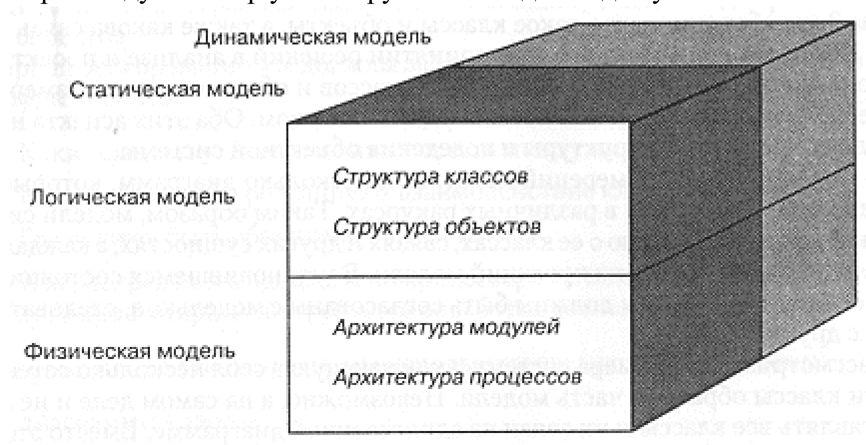


Рис. 5-1. Объектные модели

деталей отделки. Глупо было бы заранее указывать на чертеже трехмерные координаты этого выключателя (конечно, если это не является функционально важной деталью для заказчика: может быть, все члены его семьи имеют рост намного выше или ниже среднего). Если разработчики большой программной системы - высококвалифицированные специалисты и уже сработались друг с другом, им будет достаточно и грубых набросков, хотя и в этом случае они должны оставить свое творческое наследие эксплуатационщикам в удобоваримом виде. Если же разработчики неопытны, отделены друг от друга во времени или пространстве или если так установлено контрактом, то на протяжении всего процесса проектирования необходимы более детальные эскизы проекта. Система обозначений, которую мы представляем в этой главе, учитывает эти ситуации.

Различные языки программирования описывают одни и те же понятия различно. Наша система обозначений не зависит от конкретного языка. Конечно, некоторые ее элементы могут не иметь аналогов в языке, на котором будет написана проектируемая система. В этом случае просто не надо ими пользоваться. Например, в Smalltalk не бывает свободных подпрограмм, следовательно, в проект нет смысла закладывать утилиты классов. Аналогично, C++ не поддерживает метаклассы, следовательно, этот элемент должен быть исключен. Но нет ничего предосудительного в подгонке обозначений под конструкции выбранного языка. Например, в CLOS операции можно снабдить специальными пометками, чтобы определить методы

:before, **:after** и **:around**. Подобным же образом, в системе автоматического проектирования для C++ вместо спецификаций класса можно пользоваться прямо заголовочными файлами.

Цель этой главы - определить синтаксис и семантику наших обозначений для объектно-ориентированного анализа и проектирования. В качестве примера мы будем использовать гидропонную теплицу, которая рассматривалась в главе 2. В настоящей главе не обсуждается, как, собственно, были получены представленные в примерах решения: это задача главы 6. В главе 7 обсуждаются практические аспекты этого процесса, а в главах 8-12 система обозначений демонстрируется в деле на серии примеров разработки приложений.

Модели и ракурсы

В главе 3 мы объяснили, что такое классы и объекты, а также какова связь между ними. Как показано на рис. 5-1, при принятии решений в анализе и проектировании полезно рассмотреть взаимодействие классов и объектов в двух измерениях: логическом/физическом и статическом/динамическом. Оба этих аспекта необходимы для определения структуры и поведения объектной системы.

В каждом из двух измерений мы строим несколько диаграмм, которые дают нам вид моделей системы в различных ракурсах. Таким образом, модели системы содержат всю информацию о ее классах, связях и других сущностях, а каждая диаграмма представляет одну из проекций модели. В установившемся состоянии проекта, все такие диаграммы должны быть согласованы с моделью, а, следовательно, и друг с другом.

Рассмотрим для примера систему, включающую в себя несколько сотен классов; эти классы образуют часть модели. Невозможно, а на самом деле и не нужно представлять все классы и их связи на единственной диаграмме. Вместо этого мы можем описать модель в нескольких диаграммах классов, каждая из которых представляет только один ее ракурс. Одна диаграмма может показывать структуру наследования некоторых ключевых классов; другая - транзитивное замыкание множества всех классов, используемых конкретным классом. Когда модель "устоится" (придет в *устойчивое состояние*), все такие диаграммы становятся семантически согласованными друг с другом и с моделью. Например, если по данному сценарию (который мы описываем на диаграмме объектов), объект **A** посылает сообщение **M** объекту **B**, то M должно быть прямо или косвенно определено в классе **B**. В соответствующей диаграмме классов должна быть надлежащая связь между классами **A** и **B**, так, чтобы экземпляры класса A действительно могли посылать сообщение **M**.

Для простоты на диаграммах все сущности с одинаковыми именами в одной области видимости рассматриваются как ссылки на одинаковые персонажи системы. Исключением из этого правила могут быть только операции, имена которых могут быть перегружены.

Чтобы различать диаграммы, мы должны дать им имена, которые отражали бы их предмет и назначение. Можно снабдить диаграммы и другими комментариями или метками, которые мы вскоре опишем; эти комментарии, как правило, не имеют дополнительной семантики.

Логическая и физическая модели

Логическое представление описывает перечень и смысл ключевых абстракций и механизмов, которые формируют предметную область или определяют архитектуру системы. Физическая модель определяет конкретную программно-аппаратную платформу, на которой реализована система.

При анализе мы должны задать следующие вопросы:

- Каково требуемое поведение системы?
- Каковы роли и обязанности объектов по поддержанию этого поведения?

Как было отмечено в предыдущей главе, чтобы выразить наши решения о поведении системы мы пользуемся сценариями. В логической модели важнейшим средством для описания сценариев служат диаграммы объектов. При анализе могут быть полезны диаграммы классов, позволяющие увидеть общие роли и обязанности объектов.

При проектировании мы должны задать следующие вопросы относительно архитектуры системы:

- Какие существуют классы и какие есть между ними связи?
- Какие механизмы регулируют взаимодействие классов?
- Где должен быть объявлен каждый класс?
- Как распределить процессы по процессорам и как организовать работу каждого процессора, если требуется обработка нескольких процессов?

Чтобы ответить на эти вопросы, мы используем следующие диаграммы:

- диаграммы классов
- диаграммы объектов
- диаграммы модулей
- диаграммы процессов.

Статическая и динамическая модели

Четыре введенные нами типа диаграмм являются по большей части статическими. Но практически во всех системах происходят события: объекты рождаются и уничтожаются, посылают друг другу сообщения, причем в определенном порядке, внешние события вызывают операции объектов. Не удивительно, что описание динамических событий на статическом носителе, например, на листе бумаги, будет трудной задачей, но эта же трудность встречается фактически во всех областях науки. В объектно-ориентированном проектировании мы отражаем динамическую семантику двумя дополнительными диаграммами:

диаграммами переходов из одного состояния в другое

диаграммами взаимодействия.

Каждый класс может иметь собственную диаграмму переходов, которая показывает, как объект класса переходит из состояния в состояние под воздействием событий. По диаграмме объектов, имея сценарий, можно построить диаграмму взаимодействий, чтобы показать порядок передачи сообщений.

Инструменты проектирования

Плохой разработчик, имея систему автоматического проектирования, сможет создать своего программного монстра за более короткий срок чем раньше. Великие проекты создаются великими проектировщиками, а не инструментами. Инструменты проектирования дают возможность проявиться индивидуальности, освобождают ее, чтобы она могла сосредоточиться исключительно на творческих задачах проектирования и анализа. Существуют вещи, которые автоматизированные системы проектирования могут делать хорошо, и есть вещи, которые они вообще не умеют. Например, если мы используем диаграмму объектов, чтобы показать сценарий с сообщением, посылаемым от одного объекта другому, автоматизированная система проектирования может гарантировать, что посылаемое сообщение будет в протоколе объекта; это пример проверки совместимости. Если мы введем инвариант, например, такой: "существует не более трех экземпляров данного

класса", то мы ожидаем, что наш инструмент проследит за соблюдением данного требования; это пример проверки ограничения. Кроме того, система может оповестить вас, если какие-либо объявленные классы или методы не используются в проекте (проверка на полноту). Наконец, более сложная автоматическая система проектирования может определить длительность конкретной операции или достижимость состояния на диаграмме состояний; это пример автоматического анализа. Но, с другой стороны, никакая автоматическая система не сможет выявить новый класс, чтобы упростить вашу систему классов. Мы, конечно, можем попытаться создать такой инструмент, используя экспертные системы, но для этого понадобятся, во-первых, эксперты как в области объектно-ориентированного программирования, так и в предметной области, а во-вторых, четко сформулированные правила классификации и много здравого смысла. Мы не ожидаем появления таких средств в ближайшем будущем. В то же время, у нас есть вполне реальные проекты, которыми стоит заняться.

5.2. Диаграммы классов

Существенное: классы и отношения между ними

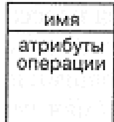
Диаграмма классов показывает классы и их отношения, тем самым представляя логический аспект проекта. Отдельная диаграмма классов представляет определенный ракурс структуры классов. На стадии анализа мы используем диаграммы классов, чтобы выделить общие роли и обязанности сущностей, обеспечивающих требуемое поведение системы. На стадии проектирования мы пользуемся диаграммой классов, чтобы передать структуру классов, формирующих архитектуру системы.

Два главных элемента диаграммы классов - это классы и их основные отношения.

Классы. На рис. 5-2 показано обозначение для представления класса на диаграмме. Класс обычно представляют аморфным объектом, вроде облака..[1]

---

[1] Выбор графических обозначении - это трудная задача. Требуется осторожно балансировать между выразительностью и простотой, так что проектирование значков - это в большой степени искусство, а не наука. Мы взяли облачко из материалов корпорации Intel, документировавшей свою оригинальную объектно-ориентированную архитектуру iAPX432 [б]. Форма этого образа намекает на расплывчатость границ абстракции, от которых не ожидается гладкости и простоты. Пунктирный контур символизирует то, что клиенты оперируют обычно с экземплярами этого класса, а не с самим классом. Можно заменить эту форму прямоугольником, как сделал Румбах [7]:



Однако, хотя прямоугольник проще рисовать, этот символ слишком часто используется в разных ситуациях и, следовательно, не вызывает ассоциаций. Кроме того, принятое Румбахом обозначение классов обычными прямоугольниками, а объектов - прямоугольниками с закругленными углами конфликтует с другими элементами его нотации (прямоугольники для актеров в диаграммах потоков данных и закругленные прямоугольники для состояний в диаграммах переходов). Облачко более удобно и для расположения пометок, которые, как мы увидим дальше, потребуются для абстрактных и параметризованных классов, и поэтому оно предпочтительнее в диаграммах классов и объектов. Аргумент простоты рисования прямоугольников спорен при использовании автоматизированной поддержки системы обозначений. Но чтобы сохранить возможность простого рисования и подчеркнуть

Каждый класс должен иметь имя; если имя слишком длинно, его можно сократить или увеличить сам значок на диаграмме. Имя каждого класса должно быть уникально в содержащей его категории. Для некоторых языков, в особенности -



Рис. 5-2. Значок класса

для C++ и Smalltalk, мы должны требовать, чтобы каждый класс имел имя, уникальное в системе.

На некоторых значках классов полезно перечислять несколько атрибутов и операций класса. "На некоторых", потому что для большинства тривиальных классов это хлопотно и не нужно. Атрибуты и операции на диаграмме представляют прообраз полной спецификации класса, в которой объявляются все его элементы. Если мы хотим увидеть на диаграмме больше атрибутов класса, мы можем увеличить значок; если мы совсем не хотим их видеть - мы удаляем разделяющую черту и пишем только имя класса.

Как мы описывали в главе 3, атрибут обозначает часть составного объекта, или агрегата. Атрибуты используются в анализе и проектировании для выражения отдельных свойств класса.[2] Мы используем следующий не зависящий от языка синтаксис, в котором атрибут может обозначаться именем или классом, или и тем и другим, и, возможно, иметь значение по умолчанию:

- **A** только имя атрибута;
- **:C** только класс;
- **A:C** имя и класс;
- **A:C=E** имя, класс и значение по умолчанию.

Имя атрибута должно быть недвусмысленно в контексте класса. В главе 3 говорилось, что операция - это услуга, предоставляемая классом. Операции обычно изображаются внутри значка класса только своим именем. Чтобы отличать их от атрибутов, к их именам добавляются скобки. Иногда полезно указать полную сигнатуру операции:

- **N()** только имя операции;
- **RN(Аргументы)** класс возвращаемого значения **(R)**, имя и формальные параметры (если есть).

Имена операций должны пониматься в контексте класса однозначно в соответствии с правилами перегрузки операций выбранного языка реализации.

Общий принцип системы обозначений: синтаксис элементов, таких, как атрибуты и операции, может быть приспособлен к синтаксису выбранного языка программирования. Например, на C++ мы можем объявить некоторые атрибуты как статические, или некоторые операции как виртуальные или чисто виртуальные;[3] в CLOS мы можем пометить операцию как метод **:around**. В любом случае мы пользуемся спецификой синтаксиса данного языка, чтобы обозначить детали. Как описывалось в главе 3, абстрактный

связь с методом Румбаха, мы оставляем его обозначения классов и объектов в качестве допустимой альтернативы.

[2] Точнее, атрибут эквивалентен отношению агрегации с физическим включением, метка которого совпадает с именем атрибута, а мощность равна в точности единице.

[3] В C++ члены, общие для всех объектов класса, объявляются статическими', виртуальной называют полиморфную операцию; чисто виртуальной называют операцию, за реализацию которой отвечает подкласс.

класс - это класс, который не может иметь экземпляров. Так как абстрактные классы очень важны для проектирования хорошей структуры классов, мы вводим для них специальный значок треугольной формы с буквой A в середине, помещаемый внутрь значка класса (рис. 5-3). Общий принцип: украшения представляют вторичную информацию о некой сущности в системе. Все подобные типы украшений имеют такой же вид вложенного треугольника.

Отношения между классами. Классы редко бывают изолированы; напротив, как объяснялось в главе 3, они вступают в отношения друг с другом. Виды отношений показаны на рис. 5-4: ассоциация, наследование, агрегация (has) и использование. При изображении конкретной связи ей можно сопоставить текстовую пометку, документирующую имя этой связи или подсказывающую ее роль. Имя связи не обязано быть глобальным, но должно быть уникально в своем контексте.

Значок ассоциации соединяет два класса и означает наличие семантической связи между ними. Ассоциации часто отмечаются существительными, например **`Employment`** (место работы), описывающими природу связи. Класс может иметь ассоциацию с самим собой (так называемая рефлексивная ассоциация). Одна пара классов может иметь более одной ассоциативной связи. Возле значка ассоциации вы можете указать ее мощность (см. главу 3), используя синтаксис следующих примеров:

- `1` В точности одна связь
- `N` Неограниченное число (0 или больше)
- `0..N` Ноль или больше
- `1..N` Одна или больше



Рис. 5-3. Значок абстрактного класса

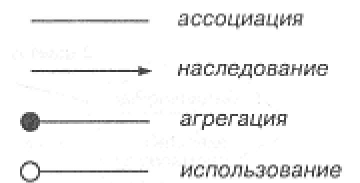


Рис. 5-4. Значки отношений между классами

- `0..1` Ноль или одна
- `3..7` Указанный интервал
- `1..3, 7` Указанный интервал или точное число

Обозначение мощности пишется у конца линии ассоциации и означает число связей между каждым экземпляром класса в начале линии с экземплярами класса в ее конце. Если мощность явно не указана, то подразумевается, что она не определена.

Обозначения оставшихся трех типов связи уточняют рисунок ассоциации дополнительными пометками. Это удобно, так как в процессе разработки проекта связи имеют тенденцию уточняться. Сначала мы заявляем о семантической связи между двумя классами, а потом, после принятия тактических решений об истинных их отношениях, уточняем эту связь как наследование, агрегацию или использование.

Значок наследования, представляющего отношение "общее/частное", выглядит как значок ассоциации со стрелкой, которая указывает от подкласса к суперклассу. В соответствии с правилами выбранного языка реализации, подкласс наследует структуру и поведение своего суперкласса. Класс может иметь один (одиночное наследование), или несколько (множественное наследование) суперклассов. Конфликты имен между суперклассами разрешаются в соответствии с правилами выбранного языка. Как правило, циклы в наследовании запрещаются. К наследованию значок мощности не приписывается.

Значок *агрегации* обозначает отношение "целое/часть" (связь "has") и получается из значка ассоциации добавлением закрашенного кружка на конце, обозначающем агрегат. Экземпляры класса на другом конце стрелки будут в каком-то смысле частями экземпляров класса-агрегата. Разрешается рефлексивная и циклическая агрегация. Агрегация не требует обязательного физического включения части в целое.

Знак использования обозначает отношение "клиент/сервер" и изображается как ассоциация с пустым кружком на конце, соответствующем клиенту. Эта связь означает, что клиент нуждается в услугах сервера, то есть операции класса-клиента вызывают операции класса-сервера или имеют сигнатуру, в которой возвращаемое значение или аргументы принадлежат классу сервера.

Пример. Описанные выше значки представляют важнейшие элементы всех диаграмм классов. В совокупности они дают разработчику набор обозначений, достаточный, чтобы описать фундамент структуры классов системы.

Рис. 5-5 показывает, как описывается в этих обозначениях задача обслуживания тепличной гидропонной системы. Эта диаграмма представляет только малую часть структуры классов системы. Мы видим здесь класс **`GardeningPlan`** (план выращивания), который имеет атрибут, названный **`crop`** (посев), одну операцию-модификатор

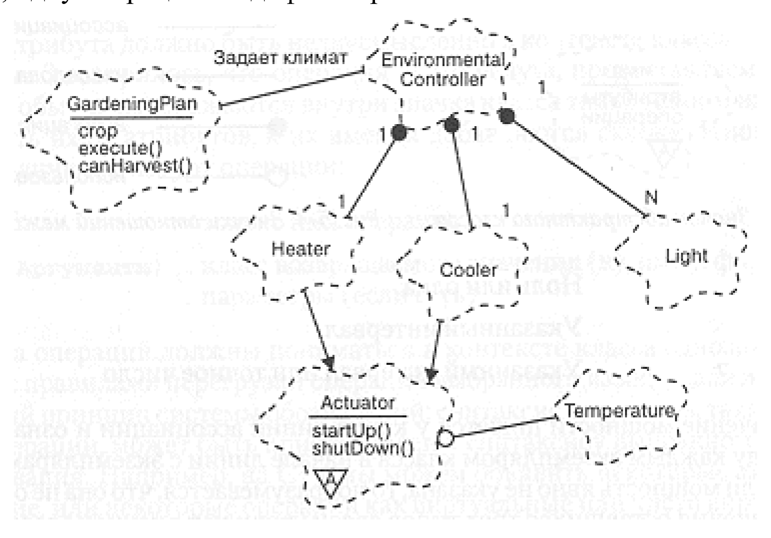


Рис. 5-5. Диаграмма классов гидропонной системы

`execute` (выполнить) и одну операцию-селектор **`canHarvest`** (можно собирать?). Имеется ассоциация между этим классом и классом **`EnvironmentalController`** (контроллер среды выращивания): экземпляры плана задают климат, который должны поддерживать экземпляры контроллера.

Эта диаграмма также показывает, что класс **`EnvironmentalController`** является агрегатом: его экземпляры содержат в

точности по одному экземпляру классов **Heater** (нагреватель) и **Cooler** (охлаждающее устройство), и любое число экземпляров класса **Light** (лампочка). Оба класса **Heater** и **Cooler** являются подклассами абстрактного запускающего процесс класса **Actuator**, который предоставляет протоколы **startUp** и **shutDown** (начать и прекратить соответственно), и который использует класс **Temperature**.

Существенное: категории классов

Как объяснялось в главе 3, класс - необходимое, но недостаточное средство декомпозиции. Когда система разрастается до дюжины классов, можно заметить группы классов, связанные внутри, и слабо зацепляющиеся с другими. Мы называем такие группы *категориями классов*.

Многие объектно-ориентированные языки не поддерживают это понятие. Следовательно, выделение обозначений для категорий классов позволяет выразить важные архитектурные элементы, которые не могли быть непосредственно записаны на языке реализации.[4]

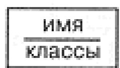


Рис. 5-6. Значок категории классов

Классы и категории классов могут сосуществовать на одной диаграмме. Верхние уровни логической архитектуры больших систем обычно описываются несколькими диаграммами, содержащими только категории классов.

Категории классов. Категории классов служат для разбиения логической модели системы. Категория классов - это агрегат, состоящий из классов и других категорий классов, в том же смысле, в котором класс - агрегат, состоящий из операций и других классов. Каждый класс системы должен "жить" в единственной категории или находиться на самом верхнем уровне системы. В отличие от класса, категория классов не имеет операций или состояний в явном виде, они содержатся в ней неявно в описаниях агрегированных классов.

На рис. 5-6 показан значок, обозначающий категорию классов. Как и для класса, для категории требуется имя, которое должно быть уникально в данной модели и отлично от имен классов.

Иногда полезно на значке категории перечислить некоторые из содержащихся в ней классов. "Некоторые", потому, что зачастую категории содержат довольно много классов, и перечислять их все было бы хлопотно, да это и не нужно. Так же, как список атрибутов и операций на значке класса, список классов в значке категории представляет сокращенный вид ее спецификации. Если мы хотим видеть на значке категории больше классов, мы можем его увеличить. Можно удалить разделяющую черту и оставить в значке только имя категории.

Категория классов представляет собой инкапсулированное пространство имен. По аналогии с квалификацией имен в C++, имя категории можно использовать для однозначной квалификации имен содержащихся в

[4] Среда программирования Smalltalk поддерживает концепцию категорий классов. Собственно это и подвигло нас на включение категорий в систему обозначений. Однако, в Smalltalk категории классов не имеют семантического содержания: они существуют только для более удобной организации библиотеки классов. В C++ категории классов связаны с концепцией компонент (Страуструп), они еще не являются чертой языка, хотя включение в него семантики пространства имен рассматривается [8]. (В настоящее время пространства имен включены в стандарт. - Примеч. ред.)

ней классов и категорий. Например, если дан класс **A** из категории **B**, то его полным именем будет **A::B**. Таким образом, как будет обсуждаться далее, для вложенных категорий квалификация имен простирается на произвольную глубину.

Некоторые классы в категории могут быть открытыми, то есть экспортироваться для использования за пределы категории. Остальные классы могут быть частью реализации, то есть не использоваться никакими классами, внешними к этой категории. Для анализа и проектирования архитектуры это различие очень важно, так как позволяет разделить обязанности между экспортируемыми классами, которые берут на себя общение с клиентами, и внутренними классами в категории, которые, собственно, выполняют работу. На самом деле, во время анализа закрытые аспекты категории классов можно опустить. По умолчанию все классы в категории определяются как открытые, если явно не указано противное. Ограничение доступа будет обсуждаться ниже.

Категория может использовать невложенные категории и классы. С другой стороны, и классы могут использовать категории. Для единообразия мы обозначаем эти экспортно-импортные отношения так же, как отношение использования между классами (см. рис. 5-4). Например, если категория **A** использует категорию **B**, это означает, что классы из **A** могут быть наследниками, или содержать экземпляры, использовать или быть еще как-то ассоциированы с классами из **B**.

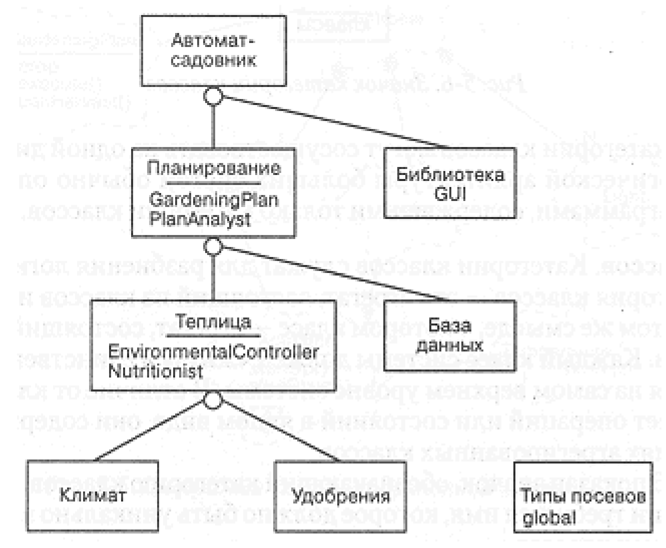


Рис. 5-7. Диаграмма классов верхнего уровня для гидропонной системы

Когда в категории слишком много общих классов, вроде базовых классов-контейнеров или других базовых классов, подобных `Object` в Smalltalk, возникают практические затруднения. Такие классы будут использоваться чуть ли не всеми другими категориями, загромождая корневой уровень диаграммы. Чтобы выйти из положения, такие категории помечаются ключевым словом **global** в левом нижнем углу значка, показывающим, что категория по умолчанию может быть использована всеми остальными.

Диаграммы классов верхнего уровня, содержащие только категории классов, представляют архитектуру системы в самом общем виде. Такие диаграммы чрезвычайно полезны для визуализации слоев и разделов системы. Слой обозначает набор категорий классов одного уровня абстракции. Таким образом, слои представляют набор категорий классов, так же как категории классов - это кластеры классов. Слои обычно нужны, чтобы изолировать верхние уровни абстракции от нижних. Разделы обозначают связанные (каким-либо образом) категории классов на разных уровнях абстракции. В этом смысле слои представляют собой горизонтальные срезы системы, а разделы - вертикальные.

Пример. На рис. 5-7 приведен пример диаграммы классов верхнего уровня для тепличного хозяйства. Это типичная многослойная система. Здесь абстракции, которые ближе к реальности (а именно активаторы и датчики климата и удобрений), располагаются на самых нижних уровнях, а абстракции, отражающие понятия пользователя, - ближе к вершине. Категория классов **`ТипыПосевов`** - глобальна, то есть ее услуги доступны всем другим категориям. На значке категории классов **`Планирование`** показаны два ее важных класса: **`GardeningPlan`** (план выращивания) с рис. 5-5 и **`PlanAnalyst`** (анализатор планов). При увеличении любой из восьми категорий классов, показанных на рисунке, обнаружатся составляющие их классы.

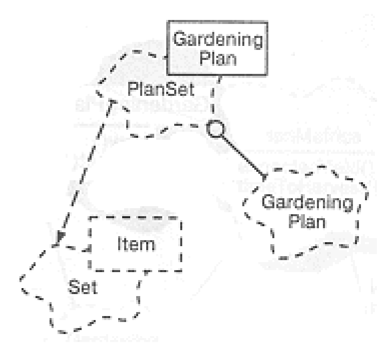


Рис. 5-8. Значок параметризованного класса

Дополнительные обозначения

До сих пор мы занимались существенной частью нашей системы обозначений.[5] Однако, чтобы передать некоторые часто встречающиеся стратегические и тактические решения, нам потребуется расширить ее. Общее правило: держаться существенных понятий и обозначений, а дополнительные применять только тогда, когда они действительно необходимы.

Параметризованные классы. В некоторых объектно-ориентированных языках программирования, в частности, C++, Eiffel и Ada можно создавать параметризованные классы. Как было сказано в главе 3, параметризованным классом называется семейство классов с общей структурой и поведением. Чтобы создать конкретный класс этого семейства, нужно подставить вместо формальных параметров фактические (процесс инстанцирования). *Конкретный класс* может порождать экземпляры.

Параметризованные классы достаточно сильно отличаются от обычных, что отмечается специальным украшением на их значках. Как показывает пример на рис. 5-8, параметризованный класс изображается значком обычного класса с пунктирным прямоугольником в правом верхнем углу, в котором указаны параметры. Инстанцированный класс изображается обычным значком класса с украшением в виде прямоугольника (со сплошной границей) с перечисленными в нем фактическими параметрами.

Связь между параметризованным классом и его инстанцированием изображается пунктирной линией, указывающей на параметризованный класс. Для получения инстанцированного класса необходим другой конкретный класс как фактический параметр (**`GardeningPlan`** в этом примере).

Параметризованный класс не может порождать экземпляры и не может использоваться сам в качестве параметра. Каждый Инстанцированный класс является новым классом, отличающимся от других конкретных классов того же семейства.

---

[5] Все существенные элементы в совокупности как раз и образуют нотацию Booch Lite.

Метаклассы. В некоторых языках, таких как Smalltalk и CLOS, есть метаклассы. Метакласс (см. главу 3) - это класс класса. В Smalltalk, например, метаклассы - это механизм поддержки переменных и операций класса (подобных статическим членам класса в C++), особенно фабрик класса (производящих операций), создающих

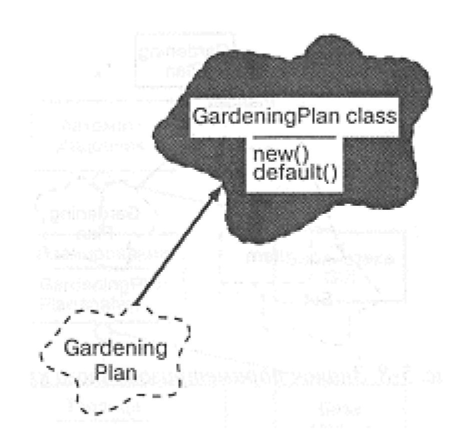


Рис. 5-9. Значок метакласса

экземпляры объектов данного класса. В CLOS метаклассы играют важную роль в возможности уточнения семантики языка [9].

Метаклассы принципиально отличаются от обычных классов, и, чтобы подчеркнуть это, их значок закрашивается серым цветом, как это сделано на рис. 5-9. Связь между классом и его метаклассом (метасвязь) имеет вид жирной стрелки, направленной от класса к его метаклассу. Метакласс **GardeningPlan** обеспечивает методы-фабрики **new()** и **default ()**, которые создают новые экземпляры класса **GardeningPlan**.

Метакласс не имеет экземпляров, но может любым образом быть ассоциирован с другими классами.

Метасвязь имеет еще одно применение. На некоторых диаграммах классов бывает полезно указать объект, который является статическим членом некоторого класса. Чтобы показать класс этого объекта, мы можем провести метасвязь "объект/ класс". Это согласуется с предыдущим употреблением: связь между некоторой сущностью (объектом или классом) и ее классом.

Утилиты классов. Благодаря своему происхождению, гибридные языки, такие как C++, Object Pascal и CLOS, позволяют разработчику применять как процедурный, так и объектно-ориентированный стиль программирования. Это контрастирует со Smalltalk, который целиком организован вокруг классов. В гибридном языке есть возможность описать функцию-не-член, называемую также *свободной подпрограммой*. Свободные подпрограммы часто возникают во время анализа и проектирования на границе объектно-ориентированной системы и ее процедурного интерфейса с внешним миром.

Утилиты классов употребляются одним из двух способов. Во-первых, утилиты класса могут содержать одну или несколько свободных подпрограмм, и тогда следует просто перечислить логические группы таких функций-не-членов. Во-вторых, утилиты класса могут обозначать класс, имеющий только переменные (и операции) класса (в C++ это означало бы класс только со статическими элементами[6]). Таким классам нет смысла иметь экземпляры, потому что все экземпляры будут находиться в одном и том же состоянии. Такой класс сам выступает в роли своего единственного экземпляра.

---

[6] Программирующие на Smalltalk часто используют идиому утилит, чтобы достичь того же эффекта.

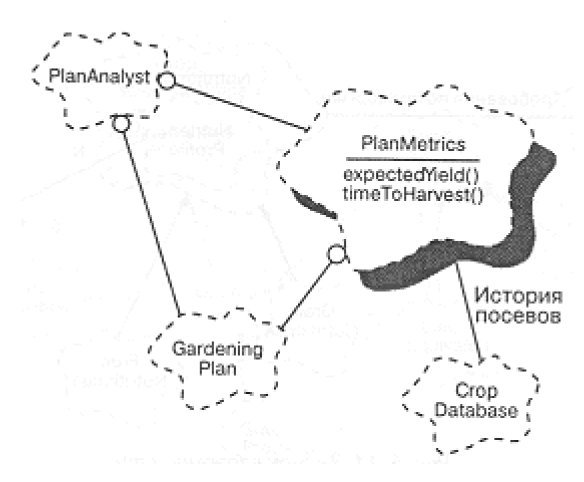


Рис. 5-10. Значок утилиты классов

Как показано на рис. 5-10, утилита классов обозначается обычным значком класса с украшением в виде тени. В этом примере утилита классов **PlanMetrics** (параметры плана) предоставляет две важные операции: **expectedYield** (ожидаемый урожай) и **timeToHarvest** (время сбора урожая). Утилита обеспечивает эти две операции на основе услуг, предоставляемых классами нижнего уровня - **GardeningPlan** (план) и **CropDatabase** (база данных об урожае). Как показывает диаграмма, **PlanMetrics** зависит от **CropDatabase**: получает от нее информацию об истории посевов. В свою очередь, класс **PlanAnalyst** использует услуги **PlanHetrics**.

Рис. 5-10 иллюстрирует обычное использование утилит классов: здесь утилита предоставляет услуги, основанные на двух независимых абстракциях нижнего уровня. Вместо того, чтобы ассоциировать эти операции с классами высшего уровня, таких как **PlanAnalyst**, мы решили собрать их в утилиту классов и добились четкого разделения обязанностей между этими простыми процедурными средствами и более изощренной абстракцией класса-анализатора **PlanAnalyst**. Кроме того, включение свободных подпрограмм в одну логическую структуру повышает шансы на их повторное использование, обеспечивая более точное разбиение абстракции.

Связь классов с утилитой может быть отношением использования, но не наследования или агрегирования. В свою очередь, утилита класса может вступать в отношение использования с другими классами и содержать их статические экземпляры, но не может от них наследовать.

Подобно классам, утилиты могут быть параметризованы и инстанцированы. Для обозначения параметризованных утилит используются такие же украшения, как и для параметризованных классов (см. рис. 5-8). Аналогично, для обозначения связи между параметризованной утилитой класса и ее конкретизацией мы используем то же обозначение, что и для инстанцирования параметризованных классов.

Вложенность. Классы могут быть физически вложены в другие классы, а категории классов - в другие категории и т. д. Обычно это нужно для того, чтобы ограничить видимость имен. Вложение соответствует объявлению вложенной сущности в окружающем ее контексте. Мы изображаем вложенность физически вложенным значком; на рис. 5-11 полное имя вложенного класса - **Nutritionist::NutrientProfile**.

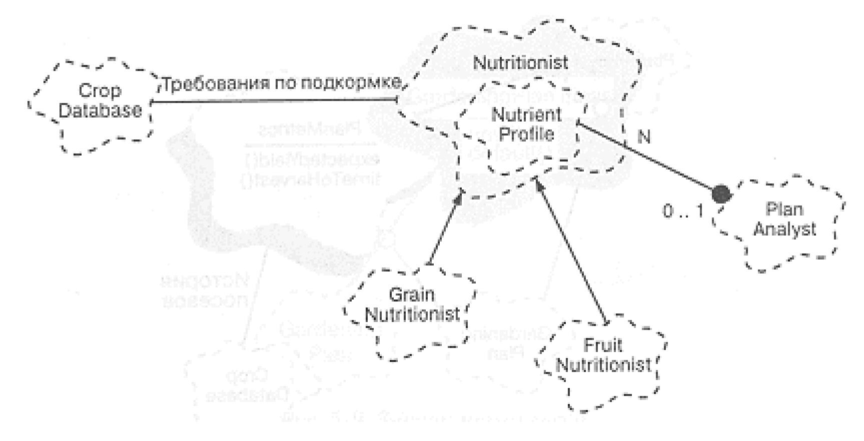


Рис. 5-11. Значок вложенности

В соответствии с правилами выбранного языка реализации, классы могут содержать экземпляры вложенного класса или использовать его. Языки обычно не допускают наследования от вложенного класса.

Обычно вложение классов является тактическим решением проектировщика, а вложение категорий классов - типично стратегическое архитектурное решение. В обоих случаях необходимость в использовании вложения на глубину более одного-двух уровней встречается крайне редко.

Управление экспортом. Все основные языки объектно-ориентированного программирования позволяют четко разделить интерфейс класса и его реализацию. Кроме того, как описано в главе 3, большинство из них позволяет разработчику определить более детально доступ к интерфейсу класса.

Например, в C++ элементы класса бывают открытыми (доступны всем клиентам), защищенными (доступны только подклассам, друзьям и самому классу) и закрытыми (доступны только самому классу и его друзьям). Кроме того, некоторые элементы могут быть частью реализации класса и тем самым быть недоступными даже друзьям этого класса.[7] В Ada элементы класса могут быть открытыми или закрытыми. В Smalltalk все переменные экземпляров по умолчанию закрытые, а все операции - открытые. Доступ предоставляется самим классом и только явно: клиент ничего не может получить насильно.

Мы изображаем способ доступа следующими украшениями связи:

•**<нет украшения>**	открытый (по умолчанию)
•|	защищенный
•||	закрытый
•|||	реализация

Мы ставим их как "засечки" на линии связи у источника. Например, на рис. 5-12 показано, что класс **GrainCrop** множественно наследует от классов **Crop** (посев) (открытый суперкласс) и **FoodItem** (пища) (защищенный суперкласс).

[7] Например, объект или класс, описанный в .cpp-файле, доступен только функциям-членам, реализованным в том же файле.

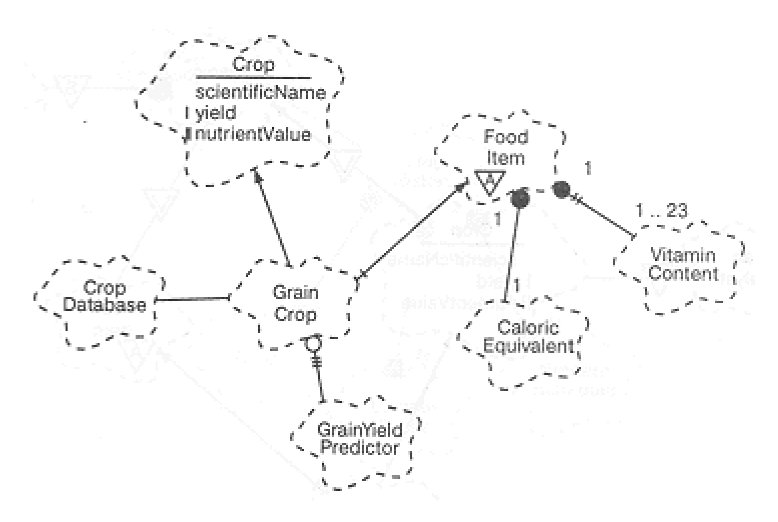


Рис. 5-12. Значок управления доступом

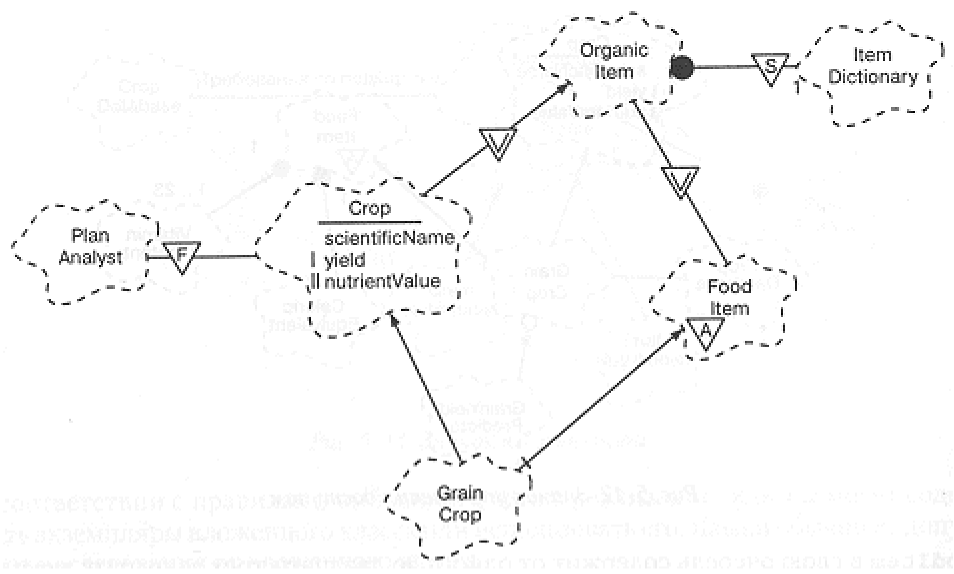
FoodItem в свою очередь содержит от одного до двадцати трех закрытых экземпляров класса **VitaminContent** (содержание витаминов) и один открытый экземпляр класса **CaloricEquivalent** (калорийность). Заметим, что **CaloricEquivalent** мог бы быть записан как атрибут класса **FoodItem**, так как атрибуты эквивалентны агрегации, мощность которой равна 1:1. Кроме того, мы видим, что класс **GrainCrop** (посев зерновых) использует класс **GrainYieldPredictor** (предсказатель урожая зерновых) как часть своей реализации. Это обычно означает, что некоторый метод класса **GrainCrop** использует услуги, предоставляемые классом **GrainYieldPredictor**.

Кроме уже рассмотренных в этом примере случаев, обычная ассоциация так же может быть украшена символами доступа. Метасвязь (связь между инстанци-рованным классом и его метаклассом) не может получить таких украшений.

Символы ограничения доступа можно применять к вложенности во всех ее формах. На обозначении класса мы можем указать доступ к атрибутам, операциям или вложенным классам, добавив символ ограничения доступа в качестве префикса к имени. Например, на рис. 5-12 показано, что класс **Crop** имеет один открытый атрибут **scientificName** (ботаническое название), один защищенный - **yield** (урожай), и один закрытый - **nutrientValue** (количество удобрения). Такие же обозначения используются для вложенных классов или категорий классов. По умолчанию все вложенные классы и категории являются открытыми, но мы можем указать ограниченный доступ соответствующей меткой.

Типы отношении. В некоторых языках встречаются настолько всепроникающие типы отношений, с настолько фундаментальной семантикой, что было бы оправдано введение новых символов. В C++, например, имеется три таких конструкции:

- **static** переменная (или функция) класса;
- **virtual** совместно используемый базовый класс в ромбовидной структуре наследования;
- **friend** класс, которому даны права доступа к закрытым и защищенным элементам другого класса.



Логично использовать для них такое же украшение в виде треугольного значка, как и для абстрактного класса, но с символами **S**,**V** или **F** соответственно.

Рассмотрим пример на рис. 5-13, который представляет другой ракурс классов, показанных на предыдущем рисунке. Мы видим, что базовый класс **OrganicItem** (органический компонент) содержит один экземпляр класса **ItemDictionary** (словарь компонентов) и что этот экземпляр содержится самим классом, а не его экземплярами (то есть он является общим для всех экземпляров). В общем случае мы указываем обозначение **static** на одном из концов ассоциации или на конце связи агрегации.

Рассматривая класс **GrainCrop**, мы видим, что структура наследования приобретает ромбовидную форму (связи наследования, разветвившись, сходятся). По умолчанию, в C++ ромбовидная форма структуры наследования ведет к тому, что в классах-листьях дублируются структуры базового, дважды унаследованного класса. Чтобы класс **GrainCrop** получил единственную копию дважды унаследованных структур класса **OrganicItem**, мы должны применить виртуальное наследование, как показано на рисунке. Мы можем добавлять украшение виртуальной связи только к наследованию.

Значок дружественности можно присоединить к любому типу связи, расположив значок ближе к серверу, подразумевая, что сервер считает клиента своим другом. Например, на рис. 5-13 класс **PlanAnalyst** дружит с классом **Crop**, а, следовательно, имеет доступ к его закрытыми и защищенным элементам, включая оба атрибута **yield** и **scientificName**.

Физическое содержание. Как показано в главе 3, отношение агрегации является специальным случаем ассоциации. Агрегация обозначает иерархию "целое/часть" и предполагает, что по агрегату можно найти его части. Иерархия "целое/часть" не означает обязательного физического содержания: профсоюз имеет членов, но это не означает, что он владеет ими. С другой стороны, отдельная запись о посеве

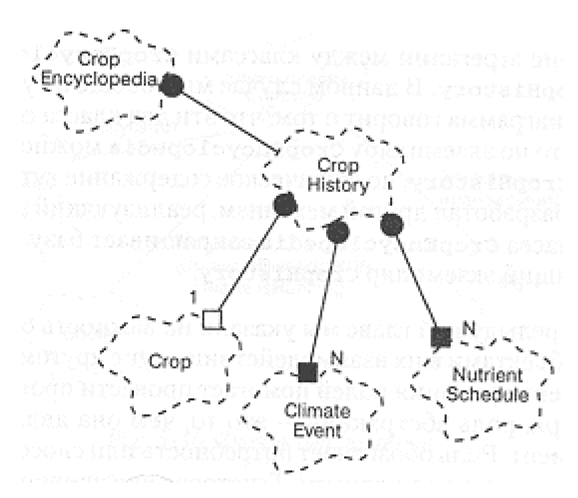


Рис. 5-14. Физическое содержание

именно физически содержит в себе соответствующую информацию, такую, как имя посева, урожай и график подкормки.

Агрегация обычно выявляется при анализе и проектировании; уточнение ее как физического содержания является детализирующим, тактическим решением. Однако, распознать этот случай важно, во-первых, для правильного определения конструкторов и деструкторов классов, входящих в агрегацию, и, во-вторых, для генерации и последовательного исправления кода.

Физическое содержание отмечается на диаграмме украшением на конце линии, обозначающей агрегацию; отсутствие этого украшения означает, что решение о физическом содержании не определено. В гибридных языках мы различаем два типа содержания:

- по значению целое физически содержит часть
- по ссылке целое физически содержит указатель или ссылку на часть.

В чисто объектно-ориентированных языках, в особенности в Smalltalk, физическое содержание бывает только по ссылке.

Чтобы отличить физическое присутствие объекта от ссылки на него, мы используем закрашенный квадратик для обозначения агрегации по значению и пустой квадратик - для агрегации по ссылке. Как будет обсуждаться позже, этот стиль украшений согласуется с соответствующей семантикой на диаграммах объектов.

Рассмотрим пример, приведенный на рис. 5-14. Мы видим, что экземпляры класса **`CropHistory`** (история посева) физически содержат несколько экземпляров классов **`NutrientSchedule`** (график внесения удобрений) и **`ClimateEvent`** (климатическое событие). Физическое содержание частей агрегации по значению означает, что их создание или уничтожение происходит при создании или уничтожении самого агрегата. Таким образом, агрегация по значению гарантирует, что время жизни агрегата совпадает с временем жизни его частей. В противоположность этому, каждый экземпляр класса **`CropHistory`** обладает только ссылкой или указателем на один экземпляр класса **`Crop`**. Это означает, что времена жизни этих двух объектов независимы, хотя и здесь один является физической частью другого. Еще один случай - отношение агрегации между классами **`CropEncyclopedia`** (энциклопедия посевов) и **`CropHistory`**. В данном случае мы вообще не упоминаем физическое содержание. Диаграмма говорит о том, что эти два класса состоят в отношении "целое/часть", и что по экземпляру **`CropEncyclopedia`** можно найти соответствующий экземпляр **`CropHistory`**, но физическое содержание тут ни при чем. Вместо этого может быть разработан другой механизм, реализующий эту ассоциацию. Например,

объект класса **CropEncyclopedia** запрашивает базу данных, и получает ссылку на подходящий экземпляр **CropHistory**.

Роли и ключи. В предыдущей главе мы указали на важность описания различных ролей, играемых объектами в их взаимодействии друг с другом; в следующей главе мы изучим, как идентификация ролей помогает провести процесс анализа.

Коротко говоря, роль абстракции - это то, чем она является для внешнего мира в данный момент. Роль обозначает потребность или способность, в силу которых один класс ассоциируется с другим. Текстовое украшение, описывающее роль класса, ставится рядом с любой ассоциацией, ближе к выполняющему роль классу, как это видно на рис. 5-15. На этом рисунке классы **PlanAnalyst** (анализатор планов) и Nutritionist (агрохимик) оба являются поставщиками информации для объекта класса **CropEncyclopedia** (они оба добавляют информацию в энциклопедию), а объекты класса **PlanAnalyst** являются также и пользователями (они просматривают материал из энциклопедии). В любом случае, роль клиента определяет индивидуальное поведение и протокол, который он использует. Обратим внимание также на рефлексивную ассоциацию класса **PlanAnalyst**: мы видим, что несколько экземпляров этого класса могут сотрудничать друг с другом и при этом они используют особый протокол, отличающийся от их поведения в ассоциации, например, с классом **CropEncyclopedia**.

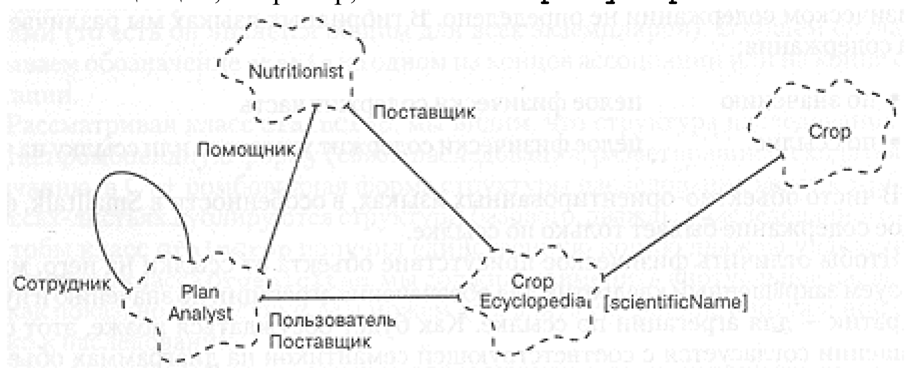


Рис. 5-15. Роли и ключи

На этом примере показана также ассоциация между классами **CropEncyclopedia** и **Crop**, но с другим типом украшения, которое представляет ключ (изображается как идентификатор в квадратных скобках). Ключ - это атрибут, значение которого уникально идентифицирует объект. В этом примере класс **CropEncyclopedia** использует атрибут **scientificName**, как ключ для поиска требуемой записи. Вообще говоря, ключ должен быть атрибутом объекта, который является частью агрегата, и ставится на дальнем конце связи-ассоциации. Возможно использование нескольких ключей, но значения ключей должны быть уникальны.

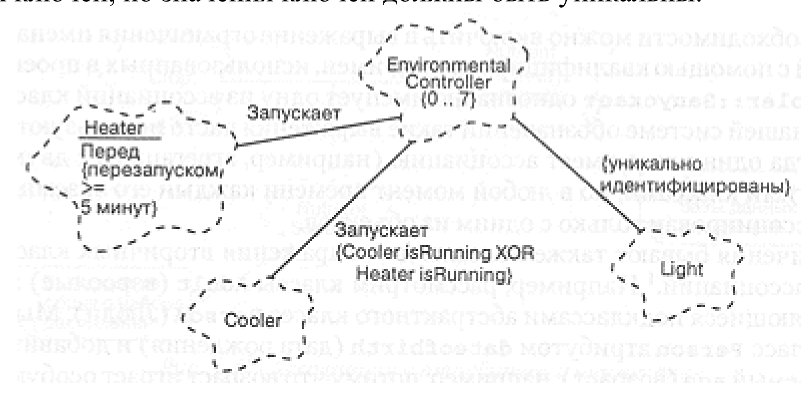


Рис. 5-16. Значок ограничения

Ограничения. Как говорилось в главе 3, ограничение - это выражение некоторого семантического условия, которое должно сохраняться. Иначе говоря, ограничение - инвариант класса или связи, который должен сохраняться, если система находится в стабильном состоянии. Подчеркнем - *в стабильном состоянии*, потому что возможны такие переходные явления, при которых меняется состояние системы в целом и система находится во внутренне рассогласованном состоянии, так что невозможно соблюсти все наложенные ограничения. Соблюдение ограничений гарантируется только в стабильном состоянии системы.

Мы используем для ограничений украшения, похожие на те, что использовались нами для обозначения ролей и ключей: помещаем заключенное в фигурные скобки выражение ограничения рядом с классом или связью, к которым оно прилагается. Ограничение присоединяется к отдельным классам, к ассоциации в целом или к ее участникам.

На рис. 5-16 мы видим, что для класса **`EnviromentalController`** наложено ограничение на мощность, постулирующее, что в системе имеется не более 7 экземпляров этого класса. При отсутствии ограничения на мощность класс может иметь сколько угодно экземпляров. Обозначение для абстрактного класса, введенное ранее, является специальным случаем ограничения (нуль экземпляров), но так как это явление очень часто встречается в иерархиях классов, оно получило собственный тип украшения (треугольник с буквой А).

Класс **`Heater`** (нагреватель) имеет ограничение другого типа. В рисунок включено требование гистерезиса в работе нагревателя: он не может быть включен, если с момента его последнего выключения прошло меньше пяти минут. Мы прилагаем это ограничение к классу **`Heater`**, считая, что контроль за его соблюдением возложен на экземпляры класса.

На этой диаграмме изображены еще два типа ограничений: ограничение на ассоциации. В ассоциации между классами **`EnvironmentalController`** и **`Light`** требуется, чтобы отдельные источники света были уникально индексированы относительно друг друга в контексте данной ассоциации. Имеется еще ограничение, наложенное на ассоциации **`EnvironmentalController`** с классами **`Heater`** и **`Cooler`**, состоящее в том, что диспетчер не может включить нагреватель и охладитель одновременно. Это ограничение прикладывается к ассоциации, а не к классам **`Heater`** и **`Cooler`**, потому что его соблюдение не может быть поручено самим нагревателям и охладителям.

При необходимости можно включить в выражение ограничения имена других ассоциаций с помощью квалифицированных имен, использованных в проекте. Например, **`Cooler`**:: запускает однозначно именует одну из ассоциаций класса-диспетчера. В нашей системе обозначений такие выражения часто используются в ситуации, когда один класс имеет ассоциацию (например, агрегацию) с двумя (или более) другими классами, но в любой момент времени каждый его экземпляр может быть ассоциирован только с одним из объектов.

Ограничения бывают также полезны для выражения вторичных классов, атрибутов и ассоциаций.[8] Например, рассмотрим классы **`Adult`** (взрослые) и **`Child`** (дети), являющиеся подклассами абстрактного класса **`Person`** (Люди). Мы можем снабдить класс **`Person`** атрибутом **`dateofbirth`** (дата рождения) и добавить атрибут, называемый **`age`** (возраст), например, потому что возраст играет особую роль в нашей модели реального мира.

---

[8] В терминологии Румбаха это называется производные сущности: для них он использует специальный значок. Нашего общего подхода к ограничениям достаточно, чтобы выразить семантику производных классов, атрибутов и ассоциации; этот подход облегчает повторное использование существующих значков и однозначное определение сущностей, от которых взяты производные.

Однако, **age** - атрибут вторичный: он может быть определен через **dateofbirth**. Таким образом, в нашей модели мы можем иметь оба атрибута, но должны указать ограничение, определяющее вывод одного из другого. Вопрос о том, какие атрибуты из каких выводятся, относится к тактике, но ограничение пригодится независимо от принятого нами решения.

Аналогично, мы могли бы иметь ассоциацию между классами **Adult** и **Child**, которая называлась бы **Parent** (родитель), а могли бы включить и ассоциацию, именуемую **Caretaker** (попечитель), если это нужно в модели (например, если моделируются официальные отношения родительства в системе социального обеспечения). Ассоциация **Caretaker** вторична: ее можно получить как следствие ассоциации Parent; мы можем указать этот инвариант как ограничение, наложенное на ассоциацию **Caretaker**.

Ассоциации с атрибутами и примечания. Последнее дополнительное понятие связано с задачей моделирования свойств ассоциаций; в системе обозначений задача решается введением элемента, который может быть приложен к любой диаграмме.

Рассмотрим пример на рис. 5-17. На нем показана ассоциация многие-ко-мно-гим между классами **Crop** и **Nutrient**. Эта ассоциация означает, что к каждому посеву применяется N (любое число) удобрений, а каждое удобрение применяется к **N** (любому числу) посевов. Класс **NutrientSchedule** является как бы свойством этого отношения многие-ко-многим: каждый его экземпляр соответствует паре из посева и удобрения. Чтобы выразить этот семантический факт, мы рисуем на диаграмме пунктирную линию от ассоциации **Crop/Nutrient** (ассоциация с атрибутом) к ее свойству - классу **NutrientSchedule** (атрибут ассоциации). Каждая уникальная ассоциация может иметь не больше одного такого атрибута и ее имя должно соответствовать имени класса-атрибута.

Идея атрибутирования ассоциаций имеет обобщение: при анализе и проектировании появляется множество временных предположений и решений; их смысл и назначение часто теряются, потому что нет подходящего места для их хранения, а хранить все в голове - дело немыслимое. Поэтому полезно ввести обозначение,

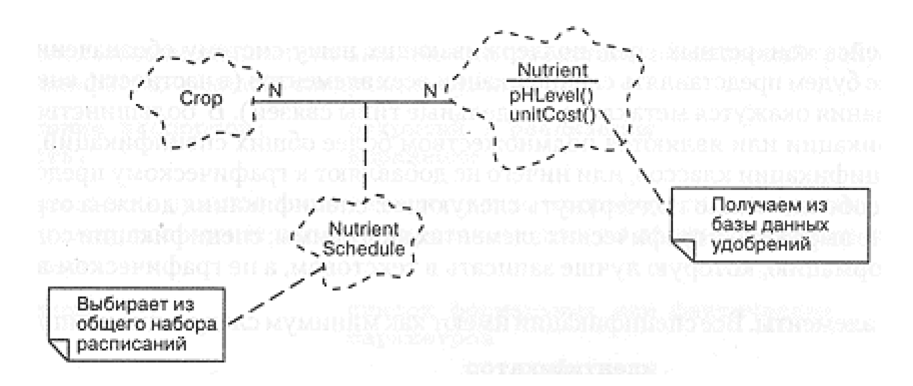


Рис. 5-17. Ассоциация с атрибутом и примечание

позволяющее добавлять произвольные текстовые примечания к любому элементу диаграммы. На рис. 5-17 имеется два таких примечания. Одно из них, приложенное к классу **NutrientSchedule**, сообщает нечто об ожидаемой уникальности его экземпляров (*Выбирает из общего набора расписаний*); другое (*Получаем из базы данных удобрений*) приложено к конкретной операции класса **Nutrient** и выражает наши пожелания к ее реализации.

Для таких примечаний мы используем значки, похожие на бумажки, и соединяем их с элементом, к которому они относятся, пунктирной линией. Примечания могут содержать любую информацию: обычный текст,

фрагменты программ или ссылки на другую документацию (все это может пригодиться при разработке инструментов проектирования). Примечания могут быть не связаны ни с каким элементом, это значит, что они относятся к самой диаграмме.[9]

Спецификации

Спецификация - это неграфическая форма, используемая для полного описания элемента системы обозначений: класса, ассоциации, отдельной операции или целой диаграммы. Просматривая диаграммы, можно относительно легко разобраться в большой системе; однако одного графического представления недостаточно: мы должны иметь некоторые пояснения к рисункам, и эту роль будут играть спецификации.

Как было сказано ранее, диаграмма - срез разрабатываемой модели системы. Спецификации же служат неграфическими обоснованиями каждого элемента обозначений. Таким образом, множество всех синтаксических и семантических фактов, нашедших свое отражение на диаграмме, должно быть подмножеством фактов, описанных в спецификации модели и согласовываться с ними. Очевидно, что важную роль в сохранении согласованности диаграмм и спецификаций может играть инструмент проектирования, поддерживающий такую систему обозначений.

В этом разделе мы рассмотрим сначала основные элементы двух важнейших спецификаций, а затем изучим их дополнительные свойства. Мы не ставим себе задачу подробного описания каждой спецификации, - оно зависит от пользовательского интерфейса конкретных сред, поддерживающих нашу систему обозначений. Мы также не будем представлять спецификации всех элементов (в частности, вне нашего внимания окажутся метакласс и отдельные типы связей). В большинстве такие спецификации или являются подмножеством более общих спецификаций, таких как спецификации классов, или ничего не добавляют к графическому представлению. Особенно важно подчеркнуть следующее: спецификация должна отражать то, что не выражено в графических элементах диаграммы; спецификации содержат ту информацию, которую лучше записать в текстовом, а не графическом виде.

`Общие элементы.` Все спецификации имеют как минимум следующие компоненты:

```
Имя:                    идентификатор
Определение:            текст
```

Уникальность имени зависит от именуемого элемента. Например, имена классов должны быть уникальны по крайней мере в содержащей их категории, тогда как имена операций имеют область видимости, локальную для содержащего их класса.

Определение - это текст, идентифицирующий представленное элементом понятие или функцию и пригодный для включения в словарь проекта (который обсуждается в следующей главе).

В каждой спецификации содержатся минимальные сведения. Конечно, используемый инструмент автоматического проектирования может вводить свои собственные графы для нужд конкретной программной среды. Однако, важно указать, что независимо от того, сколько граф включает в себя спецификация, не следует навязывать разработчику дурацкие правила, по которым он обязан заполнить все части спецификации, прежде чем приступит

[9] Значок, который мы используем, похож на обозначение примечаний во многих windows-системах, особенно следующих традициям Macintosh. Непосредственными вдохновителями нашего обозначения были предложения Гамма, Хелпа, Джонсона и Влисси-Деса[10].

к следующему этапу разработки. Обозначения должны облегчать разработку, а не создавать дополнительные трудности.

Спецификации класса. Каждый класс в модели имеет ровно одну спецификацию, в которой содержатся как минимум следующие пункты:

```
Обязанности:          текст
Атрибуты:             список атрибутов
Операции:             список операций
Ограничения:          список ограничений
```

Как говорилось в предыдущей главе, обязанности класса - это список предоставляемых им гарантий поведения. В следующей главе будет показано, как мы используем эту графу для регистрации обязанностей классов, которые мы открываем или изобретаем в процессе разработки.

Остальные пункты - атрибуты, операции, ограничения - соответствуют их графическим аналогам. Некоторые операции могут быть настолько важными, что следует снабдить их собственными спецификациями, которые мы обсудим ниже.

Перечисленные основные понятия могут быть представлены в терминах выбранного языка реализации. В частности, все эти сведения, как правило, однозначно фиксируются объявлением класса на C++ или спецификацией пакета в Ada.

Как говорилось в главе 3, часто поведение некоторых важных классов наилучшим образом выражается на языке конечного автомата, поэтому мы включим в спецификацию класса дополнительную графу:

```
Автомат:                  ссылка на автомат
```

Использование дополнительных элементов системы обозначений требует ввести в спецификацию класса следующие пункты:

```
Управление экспортом:     открытый | реализация
Мощность:                 выражение
```

Смысл этих пунктов вполне тождественен их графическим аналогам. Параметризованные и инстанцированные классы должны включать следующий пункт:

```
Параметры:                список формальных или фактических
                             параметров
```

Следующие необязательные пункты не имеют графических аналогов; они служат, чтобы указать некоторые функциональные аспекты класса:

```
Устойчивость:             мгновенный | постоянный
Параллельность:           последовательный | охраняемый |
                             синхронный | активный
Место в памяти:           выражение
```

Первое из этих свойств отражает продолжительность жизни объектов класса:

постоянная сущность - это та, чье состояние может пережить сам объект, в отличие от мгновенных, состояние которых пропадает с истечением времени жизни объекта.

Второе свойство показывает в какой степени класс может работать в многопоточной системе (см. главу 2). По умолчанию объекты - последовательные, то есть рассчитаны на один поток. Охраняемый и синхронный классы "выдерживают" несколько потоков. При этом охраняемый класс ожидает, что клиентские потоки как-то договариваются о взаимном исключении, с тем чтобы в каждый момент времени с ним работал только один из них. Синхронный класс сам обеспечивает взаимное исключение клиентов. Наконец, активный класс имеет свой поток.

Последний пункт содержит сведения об абсолютном или относительном потреблении памяти объектами этого класса. Мы можем использовать эту графу для подсчета размера класса или его экземпляров.

Спецификации операций. Для всех операций-членов классов и свободных подпрограмм наши спецификации включают следующие основные пункты:

```
Класс возвращаемого значения:  ссылка на класс
Аргументы:                     список формальных аргументов
```

Эти графы можно заполнить на выбранном языке реализации. В соответствии с правилами языка можно включить еще один пункт:

```
Квалификация:                   текст
```

В C++, например, этот пункт может содержать утверждение о том, является ли операция статической, виртуальной, чисто виртуальной или константой.

Использование дополнительных элементов обозначений требует введения дополнительной графы:

```
Доступ:           открытый | защищенный | закрытый | реализация
```

Содержание этой графы зависит от языка реализации. Например в Object Pascal все атрибуты и операции всегда открытые, в Ada операции могут быть открытыми или закрытыми, а в C++ возможны любые из четырех указанных случаев.

Использование дополнительных элементов обозначений требует также введения графы

```
Протокол:                текст
```

Эта графа происходит из практики языка Smalltalk: протокол операции не имеет семантического значения, а служит просто для именования логической совокупности операций, вроде таких, как initialize-release (инициализация-освобождение) или model access (доступ к модели).

Следующие необязательные графы не имеют графических аналогов и служат для формального описания семантики операции:

```
Предусловия:           текст | ссылка на текст программы |
ссылка на
                                          диаграмму объектов
Семантика:             текст | ссылка на текст программы |
ссылка на
                                         диаграмму объектов
Постусловия:           текст | ссылка на текст программы |
ссылка на
                          диаграмму объектов
```

```
Исключения:             список исключительных ситуаций
```

Первые три пункта могут быть заполнены в любой из перечисленных форм. Последний содержит список исключительных ситуаций, содержащий имена соответствующих классов.

Последняя серия необязательных граф служит для описания некоторых функциональных аспектов операции:

```
Параллельность:         последовательный |   охраняемый |
синхронный
Память:                 выражение
Время:                  выражение
```

Первые две аналогичны одноименным графам в спецификации класса. Третья - относительные или абсолютные оценки времени выполнения операции.

5.3. Диаграммы состояний и переходов

Существенное: состояния и переходы

Диаграмма состоянии и переходов показывает: пространство состояний данного класса; события, которые влекут переход из одного состояния в другое; действия, которые происходят при изменении состояния. Мы приспособили обозначения, использованные Харелом [11]: его работа предоставляет простой, но очень выразительный подход, который гораздо эффективнее традиционных автоматов с конечным числом состоянии.[10] Отдельная диаграмма состояний и переходов представляет определенный ракурс динамической модели отдельного класса или целой систе-



Рис. 5-18. Значок состояния

мы. Мы строим диаграммы состояний и переходов только для классов, поведение которых (управляемое событиями) для нас существенно. Мы можем также представить диаграмму состояний и переходов для управляемого событиями поведения системы в целом. Эти диаграммы используются в ходе анализа, чтобы показать динамику поведения системы, а в ходе проектирования - для выражения поведения отдельных классов или их взаимодействия.

Два основных элемента диаграммы состояний и переходов - это, естественно, состояния и переходы между ними.

Состояния. Состояние представляет собой итоговый результат поведения системы. Например, только что включенный в сеть телефон находится в начальном состоянии: его предыдущее поведение несущественно, при этом он готов к тому, чтобы позвонить или принять звонок. Если кто-нибудь поднимет трубку, телефон перейдет в состояние готовности к набору номера; в этом состоянии мы не ожидаем, что телефон зазвонит, но приготовились к беседе с одним или несколькими абонентами. Если кто-либо

[10] Мы дополнили его работу применительно к объектно-ориентированному программированию, следуя предложениям Румбаха [12] и Беара и др. [13].

наберет ваш номер, а телефон находится в начальном состоянии (трубка положена), то когда вы поднимете трубку, телефон перейдет в состояние с установленным соединением, и вы сможете поговорить со звонившим.

В любой момент времени состояние объекта определяет набор свойств (обычно статический) объекта и текущие (обычно динамические) значения этих свойств. Под "свойствами" подразумевается совокупность всех связей и атрибутов объекта. Мы можем обобщить понятие состояния так, чтобы оно было применимо и к объекту, и к классу, так как все объекты одного класса "живут" в одном пространстве состояний. Это пространство может представлять собой неопределенное, хотя конечное множество возможных (но не всегда ожидаемых или желаемых) состояний. На рис. 5-18 показано обозначение, которое мы используем для отдельного состояния.

Каждое состояние должно иметь имя; если оно оказывается слишком длинным, то его можно сократить или увеличить значок состояния. Каждое имя состояния должно быть уникально в своем классе. Состояния, ассоциированные со всей системой, глобальны, то есть видимы отовсюду, а область видимости вложенных состояний (дополнительное понятие) - ограничена соответствующей подсистемой. Все одноименные значки состояний на одной диаграмме обозначают одно и то же состояние.

На значках некоторых состояний полезно указать ассоциированные с ними действия. Как показано на рис. 5-18, действия обозначаются так же, как атрибуты и операции в значке класса. Мы можем увеличить значок, чтобы увидеть весь список действий, или, если нет необходимости указывать действия, можно удалить разделяющую линию и оставить только имя.[11] Ассоциацию действий с состояниями мы обсудим позднее.

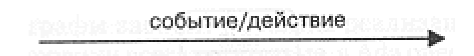


Рис. 5-19. Значок перехода из состояния в состояние

Переходы. Событием мы называем любое происшествие, которое может быть причиной изменения состояния системы. Изменение состояний называется переходом. На диаграмме переходов и состояний он изображается значком, показанным на рис. 5-19. Каждый переход соединяет два состояния. Состояние может иметь переход само в себя; обычно есть несколько различных переходов в одно и тоже состояние, но все переходы должны быть уникальны в том смысле, что ни при каких обстоятельствах не может произойти одновременно два перехода из одного состояния.

Например, в поведении гидропонной теплицы играют роль следующие события:

- Посажена новая партия семян
- Урожай созрел и готов к сбору
- Из-за плохой погоды упала температура в теплице
- Отказало охлаждающее устройство
- Наступил заданный момент времени.

Как будет рассказано в следующей главе, идентификация событий, подобных этим, позволяет определить границы поведения системы и распределить обязанности по осуществлению этого поведения между отдельными классами.

Каждое из первых четырех перечисленных выше событий, вероятно, вызывает некоторое действие - например, начало или остановку выполнения некоторого плана сельскохозяйственных работ по посеву, включение

---

[11] Для совместимости с обозначениями Харела разделяющую линию можно вообще убрать.

нагревателя или посылку сигнала тревоги технику, обслуживающему систему. Отсчет времени - это другое дело: хотя секунды и минуты не имеют значения (посевы растут, очевидно, не так быстро), наступление нового часа или суток может вызвать некоторый сигнал, например, включить/выключить лампочки и изменить температуру в теплице, чтобы имитировать смену дня и ночи, необходимую для роста растений.

Действием мы называем операцию, которая, с практической точки зрения, требует нулевого времени на выполнение. Например, включение сигнала тревоги - действие. Обычно действие означает вызов метода, порождение другого события, запуск или остановку процесса. Деятельностью мы называем операцию, требующую некоторого времени на свое выполнение. Например, нагрев воздуха в теплице - деятельность, запускаемая включением нагревателя, который может оставаться включенным неопределенное время, до тех пор, пока не будет выключен явной командой.

Модель событий, передающих сообщения, которую предложил Харел, концептуально безупречна, но ее нужно приспособить к объектному подходу. При анализе мы можем давать предварительные названия событиям и действиям, в общих чертах отражая наше понимание предметной области. Однако, отображая эти понятия на классы, мы должны предложить конкретную стратегию реализации.

Событие может быть представлено символическим именем (или именованным объектом), классом или именем некоторой операции. Например, событие **CoolerFailure** (неисправность охлаждающего устройства) может обозначать либо литерал, либо имя объекта. Мы можем придерживаться той стратегии, что все события являются символическими именами и каждый класс с поведением, управляемым событиями, имеет операцию, которая распознает эти имена и выполняет соответствующие действия. Такая стратегия часто используется в архитектурах типа *модель-представление-котроллер* (model-view-controller), которая пришла из языка Smalltalk. Для большей общности можно считать события объектами и определить иерархию классов, которые представляют собой абстракции этих событий. Например, можно определить общий класс событий **DeviceFailure** (неисправность устройства) и его специализированные подклассы, такие как **CoolerFailure** (неисправность охлаждающего устройства) и **HeaterFailure** (неисправность нагревателя). Теперь извещение о событии можно связать с экземпляром класса-листа (например, **CoolerFailure**) или более общего суперкласса (**DeviceFailure**). И если выполнение некоторого действия назначено только при возникновении события класса **CoolerFailure**, то это означает, что все другие случаи отказа устройств должны намеренно игнорироваться. С другой стороны, если выполнение действия связано с событием **DeviceFailure**, то действие должно выполняться независимо от того, на каком устройстве произошел сбой. Продолжая в том же духе, мы можем сделать так, чтобы переходы из состояния в состояние были полиморфны относительно классов событий. Наконец, можно определить событие просто как операцию, такую как **GardeningPlan:: execute ()**. Это похоже на подход, который трактует события как имена, но в отличие от него здесь не требуется явного диспетчера событий.

Для нашего метода несущественно, какая из этих стратегий выбрана для разработки, если она последовательно проводится во всей системе. Обычно в замечаниях указывается, какая стратегия использована для данного конкретного автомата.

Действие можно записывать, используя синтаксис, показанный в следующих примерах:

- **heater.startUp()** действие
- **DeviceFailure** произошло событие
- **start Heating** начать некоторую деятельность

•**stop Heating** прекратить деятельность.

Имена операций или событий должны быть уникальны в области видимости диаграммы; там, где необходимо, они могут быть квалифицированы соответствующими именами классов или объектов. В случае начала или прекращения некоторой деятельности, она может быть представлена операцией (такой, как **Actuator::shutDown ()**) или символическим именем (для событий). Когда деятельность соответствует некоторой функции системы, такой, как **harvest crop** (сбор урожая), мы обычно пользуемся символическими именами.

На каждой диаграмме состояний и переходов должно присутствовать ровно одно стартовое состояние; оно обозначается немаркированным переходом в него из специального значка, изображаемого в виде закрашенного кружка. Иногда бывает нужно указать также конечное состояние (обычно автомат, ассоциированный с классом или системой в целом, никогда не достигает конечного состояния; этот автомат просто перестает существовать после того, как содержащий его объект уничтожается). Мы обозначаем конечное состояние, рисуя немаркированный переход от него к специальному значку, изображаемому как кружок с закрашенной серединой.

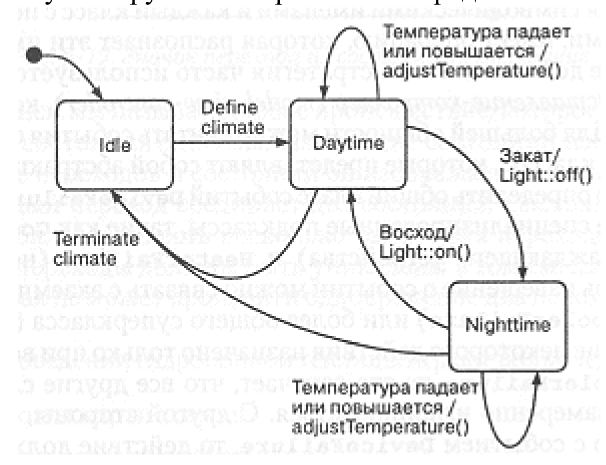


*Рис. 5-20. Диаграмма состояний и переходов для контроллера тепличной среды (**EnvironmentalController**)*

Пример. До сих пор вводились значки, описывающие существенные элементы диаграмм состояний и переходов. В совокупности они предоставляют разработчику систему обозначений, достаточную для моделирования простого конечного плоского автомата, пригодного для описания приложений с ограниченным числом состояний. Системы, имеющие много состояний или обладающие сильно запутанным событийным поведением, которое описывается переходами по условию или в результате предыдущих состояний, требуют для построения диаграмм переходов более сложных понятий.

На рис. 5-20 показан пример использования существенных обозначений. Пример опять описывает гидропонную систему. Мы видим диаграмму состояний и переходов для класса **EnvironmentalController**, впервые введенного на рис. 5-5.

На этой диаграмме все события представляются символическими именами. Мы видим, что все объекты этого класса начинают свою жизнь в начальном состоянии **Idle** (ожидание); затем они изменяют свое состояние по событию **Define climate**, для которого не предполагается явных действий (считается, что это событие, то есть ввод климатического задания, происходит только в дневное время). Дальше динамическое поведение этого класса состоит в переключении между со-стояниями **Daytime** и **Nighttime** (день и ночь); оно определяется событиями **Sunrise** и **Sunset** (восход и закат) соответственно; с этими событиями связаны действия по изменению

освещения. В обоих состояниях событие понижения или повышения температуры в теплице вызывает обратную реакцию (операция **adjustTemperature()**, которая является локальной в этом классе). Мы возвращаемся в состояние **Idle**, когда поступит событие **Terminate climate**, то есть будет отменено климатическое задание.

Дополнительные понятия

Элементы диаграмм состояний и переходов, которые мы только что описали, недостаточны для многих случаев сложных систем. По этой причине мы расширим наши обозначения, включив семантику карт состояний, предложенную Харелом.

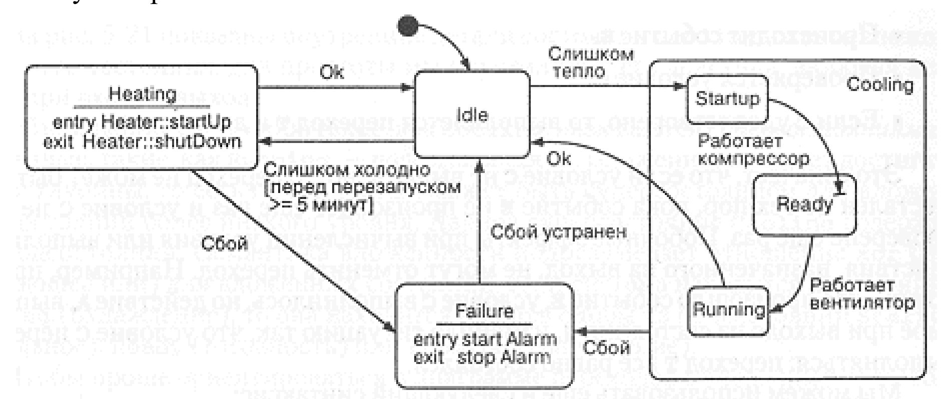


Рис. 5-21. Действия, условные переходы и вложенные состояния

Действия, ассоциированные с состояниями и условные переходы. Как показано на рис. 5-18, с состояниями могут быть ассоциированы действия. В частности, можно назначить выполнение некоторого действия на входе или выходе из состояния, при этом используется синтаксис следующих примеров:

- **entry start Alarm** — запуск процедуры при входе в состояние
- **exit shutDown()** — вызов операции при выходе из состояния.

Как и для переходов, можно назначить любое действие после ключевых слов **entry** и **exit**(вход и выход).

Деятельность можно ассоциировать с состоянием, используя синтаксис следующего примера:

- **do Cooling** — в данном состоянии заниматься этой деятельностью.

Этот синтаксис служит сокращенной записью явных указаний: "Начать деятельность при входе в состояние и окончить при выходе из него".

На рис. 5-21 мы видим пример использования этих обозначений. При входе в состояние **Heating** (нагревание) вызывается операция **Heater::startUp()**, а при выходе - операция **Heater::shutDown()**, то есть происходит запуск и остановка нагревания. При входе и выходе из состояния **Failure** (сбой), соответственно вызывается и прекращается сигнал тревоги (**Alarm**).

Рассмотрим также переход из состояния **Idle** в состояние **Heating**. Он совершается, если температура понизилась, но только в случае, если прошло больше пяти минут после того, как последний раз был выключен нагреватель. Это пример условного (или защищенного) перехода; условие представляется логическим выражением в скобках.

Вообще, каждый переход может быть ассоциирован либо с событием, либо с событием и условием. Допускаются и "переходы без события". В этом

случае переход совершается сразу после завершения действия, связанного с состоянием, причем выполняется и действие, связанное с выходом из этого состояния. Если переход условный, он состоится только в случае, если условие выполнено.

Имеет значение порядок выполнения условного перехода. Пусть имеется состояние **S**, из которого при событии **E** совершается переход **T** с условием **C** и действием **A**. Переход **T** осуществляется в такой последовательности:

- Происходит событие **E**.
- Проверяется условие **C**.
- Если **C** удовлетворено, то выполняется переход **T** и действие **A**.

Это означает, что если условие **C** не выполнено, то переход не может быть осуществлен до тех пор, пока событие **E** не произойдет еще раз и условие **C** не будет проверено еще раз. Побочные эффекты при вычислении условия или выполнении действия, назначенного на выход, не могут отменить переход. Например, предположим, что произошло событие **E**, условие **C** выполнилось, но действие **A**, выполняемое при выходе из состояния **S**, изменило ситуацию так, что условие **C** перестало выполняться: переход **T** все равно состоялся.

Мы можем использовать еще и следующий синтаксис:

- **in Cooling** выражение для текущего состояния.

Здесь используется имя состояния (которое может быть квалифицированным). Выражение истинно тогда и только тогда, когда система находится в указанном состоянии. Такие условия особенно полезны, когда некоторому внешнему состоянию нужно запустить переход по условию, связанному с некоторым вложенным состоянием.

Можно использовать в условии и выражение, налагающее ограничения по времени:

- **timeout (Heat ing, 30)** выражение ограничения по времени.

Это условие выполняется, если система более 30 секунд находилась в состоянии **Heating** и остается в нем в момент проверки. Этот тип условия употребляется в системах реального времени для "переходов без события", так как защищает систему от зависания на долгое время в одном состоянии. Это выражение можно использовать для указания нижней границы времени нахождения в данном состоянии. Если приложить временное ограничение к каждому переходу с событием, выводящим из данного состояния, это будет равнозначно требованию, что система находится в каждом состоянии как минимум время, указанное в ограничении.[12]

Что случится, если некое событие произойдет, а перейти в другое состояние нельзя либо потому, что не существует перехода для данного события, либо не выполняется условие перехода? По умолчанию это надо считать ошибкой: игнорирование событий обычно является признаком неполного анализа задачи. Вообще, для каждого состояния нужно документировать события, которые оно намеренно игнорирует.

Вложенные состояния. Возможность вложения состояний друг в друга придает глубину диаграммам переходов; эта ключевая особенность карт состояний Харела предотвращает комбинаторный взрыв в структуре состояний и переходов, который часто случается в сложных системах.

На рис. 5-21 показаны внутренние детали состояния **Cooling**, то есть вложенные в него состояния; для простоты мы опустили все его действия, включая действия при входе и выходе.

---

[12] Харел предложил "обобщенную завитушку" для обозначения двухсторонних границ по времени, но мы не будем обсуждать здесь его обобщения, так как условия исчерпания времени достаточно выразительны.

Объемлющие состояния, такие, как **Cooling**, называются суперсостояниями, а вложенные, такие, как **Running**, - подсостояниями. Вложенность может достигать любой глубины, то есть подсостояние может быть суперсостоянием для вложенных состояний более низкого уровня. Данное суперсостояние **Cooling** содержит три подсостояния. Семантика вложенности подразумевает отношение **xor** (исключающее или) для вложенных состояний: если система находится в состоянии **Cooling** (охлаждение), то она находится ровно в одном из подсостояний **Startup** (начальное), **Ready** (готовность) или **Running** (выполнение).

Чтобы проще ориентироваться в диаграмме переходов с вложенными состоя-ниями мы можем увеличить или уменьшить ее масштаб относительно выбранного состояния. При уменьшении вложенные состояния исчезают, а при увеличении проявляются. Переходы в скрытые на диаграмме подсостояния и выходы из них показываются стрелкой с черточкой, как переход в состояние **Ready** на рисунке.[13]

Переходам между состояниями разрешено начинаться и кончаться на любом уровне. Рассмотрим различные формы переходов:

- Переход между одноуровневыми состояниями (такой, как из **Failure** В **Idle** или из **Ready** в **Running**) - простейшая форма перехода; его семантика описана в предыдущем разделе.
- Можно совершить переход непосредственно в подсостояние (как из **Idle** в **Startup**), или непосредственно из подсостояния (как из **Running** в **Idle**), или одновременно и то, и другое.
- Указание перехода из суперсостояния (как из **Cooling** в **Failure** через событие **Failure**) означает, что он осуществляется из каждого подсостояния этого суперсостояния. Такой переход пронизывает все уровни до переопределения. Это упрощает диаграмму за счет удаления банальных переходов, общих для всех подсостояний.
- Указание перехода в состояние с вложенными подсостояниями (например, предыдущий переход в состояние **Failure**) подразумевает переход к его начальному подсостоянию (по умолчанию).

История. Иногда, возвращаясь к суперсостоянию, мы хотели бы попасть в то его подсостояние, где мы были последний раз. Эту семантику мы будем изображать значком истории (буква **H** (History) внутри кружка, размещенного где-нибудь внутри значка суперсостояния). Например, на рис. 5-22 мы видим развернутое изображение состояния **Failure**. В самый первый раз, когда наша система переходит в него, она принимает начальное состояние по умолчанию **Create log** (создать журнал); что обозначено непомеченным переходом из закрашенного кружка внутри объемлющего состояния; когда журнал (**log**) создан, система переходит в состояние **Log ready**. После того, как сообщение о сбое занесено в журнал, мы возвра-

---

[13] Если быть точными, то переходы **Too hot** и **ok** относительно состояния **Cooling** также должны быть показаны на рис. 5-21 с черточкой, так как это переходы между подсостояниями.

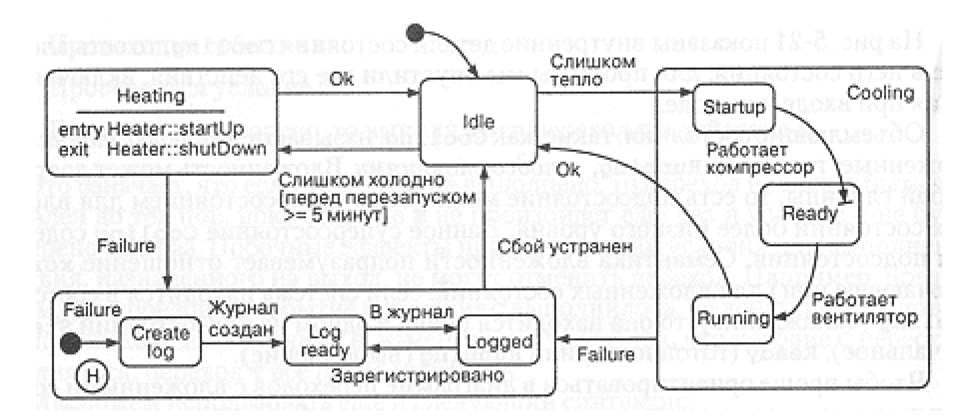


Рис. 5-22. История событий

щаемся обратно. Когда мы попадем в состояние **Failure** в следующий раз, нам не нужно будет опять создавать журнал, и мы перейдем прямо к **Log ready**, так как когда мы в последний раз выходили из состояния **Failure**, система находилась именно в этом подсостоянии.

Действие "истории" распространяется только на тот уровень, на котором она указана. Если мы хотим распространить ее действие на все нижние подуровни, то мы обозначим это, пририсовав к ее значку звездочку. Можно получить промежуточный результат, пририсовав значок истории только к отдельным подсостояниям.

Спецификации

Каждый элемент диаграммы переходов может иметь спецификацию, которая дает его полное определение. В отличие от спецификаций классов, спецификации переходов и состояний ничего не добавляют к уже описанному в этом разделе, поэтому нет необходимости обсуждать их специально.

5.4. Диаграммы объектов

Существенное: объекты и их отношения

Диаграмма объектов показывает существующие объекты и их связи в логическом проекте системы. Иначе говоря, диаграмма объектов представляет собой мгновенный снимок потока событий в некоторой конфигурации объектов. Таким образом, диаграммы объектов являются своего рода прототипами: каждая представляет взаимодействие или структурные связи, которые могут возникнуть у данного множества экземпляров классов, безотносительно к тому, какие конкретно экземпляры участвуют в этом взаимодействии. В таком смысле, отдельная диаграмма объектов есть ракурс структуры объектов системы. При анализе мы используем диаграммы объектов для показа семантики основных и второстепенных сценариев, которые отслеживают поведение системы. При проектировании мы используем диаграммы объектов для иллюстрации семантики механизмов в логическом проектировании системы. Существенные элементы диаграммы объектов - объекты и их отношения.



Рис. 5-23. Значок объекта

Объекты. На рис. 5-23 показан значок, который изображает объект на диаграмме объектов. Как и в диаграммах классов, можно провести горизонтальную линию, разделяющую текст внутри значка объекта на две части: имя объекта и его атрибуты.

Имя объекта следует синтаксису для атрибутов, и может быть либо записано в одной из трех следующих форм:

- `A` только имя объекта
- `:C` только класс объектов
- `A:C` имя объекта и класса

либо использовать синтаксис выбранного языка реализации. Если текст не умещается внутри значка, то следует или увеличить значок, или сократить текст. Если несколько значков объектов на одной диаграмме используют одно и то же неквалифицированное имя (то есть имя без указания класса), то они означают один и тот же объект. В противном случае каждый значок означает отдельный объект.[14] Если на разных диаграммах есть объекты с одинаковыми неквалифицированными именами, то это разные объекты, если только их имена не квалифицированы явно.

Смысл неквалифицированных имен зависит от контекста диаграммы объектов. Более точно: диаграммы объектов, определенные на самом верхнем уровне системы, имеют глобальную область видимости; другие диаграммы объектов могут быть определены для категорий классов, отдельных классов или отдельных методов, а, значит, иметь соответствующие области видимости. Квалифицированное имя может быть использовано при необходимости явной ссылки на глобальные объекты, переменные классов (статические элементы в C++), параметры методов, атрибуты или локально определенные объекты в той же области видимости.

Если не указать класс объекта - ни явно, использовав ранее упомянутый синтаксис, ни косвенно, через спецификацию объекта, - то класс рассматривается как анонимный, при этом нельзя провести семантическую проверку ни операций, совершаемых над объектом, ни его связей с другими объектами на диаграмме. Если же указать только класс, то анонимным считается объект. Каждый значок без имени объекта обозначает отдельный анонимный объект.

В любом случае, имя класса объекта должно быть именем настоящего класса (или любого из его суперклассов) в области видимости диаграммы, использованного для инстанцирования объекта, даже если этот класс - абстрактный. Эти правила позволяют написать сценарий, не зная точно, к каким подклассам принадлежат объекты.



Рис. 5-24. Значок связи между объектами

На значках объектов бывает полезно указать несколько их атрибутов. "Несколько" - так как значок объекта представляет только один какой-то ракурс его структуры. Синтаксис атрибутов совпадает с описанным ранее синтаксисом атрибутов класса и позволяет указать выражение, используемое по умолчанию. Имена атрибутов объектов должны соответствовать атрибутам, определенным в классе объекта, или в любом из его суперклассов. Синтаксис имен атрибутов может быть приспособлен к синтаксису языка реализации.

---

[14] На одной диаграмме могут присутствовать значки объектов с одинаковыми неквалифицированными именами, но относящиеся к разным классам, в том случае, если эти классы имеют общего предшественника. Это позволяет представить распространение операций от подкласса к суперклассу и наоборот.

Диаграмма объектов может также включать значки, обозначающие утилиты классов и метаклассы: эти понятия подобны объектам, так как они могут действовать как объекты, и с ними можно оперировать как с объектами.

Отношения между объектами. Как говорилось в главе 3, объекты взаимодействуют с другими объектами через связи, обозначение которых показано на рис. 5-24. Подобно тому, как объект является экземпляром класса, связь является экземпляром ассоциации.

Связь между двумя объектами (включая утилиты классов и метаклассы) может существовать тогда и только тогда, когда существует ассоциация между соответствующими классами. Ассоциация между классами может проявляться различными способами, например, как простая ассоциация, отношение наследования или отношение включения. Следовательно, существование ассоциаций между двумя классами означает существование коммуникации (то есть канала связи) между их экземплярами, по которой объекты могут посылать друг другу сообщения. Все классы неявно имеют ассоциации сами с собой и, следовательно, объект может послать сообщение сам себе.

Пусть имеются объекты **`A`** и **`B`** и связь **`L`** между ними. Тогда **`A`** может вызвать любую операцию, имеющуюся в классе **`B`** и доступную **`A`**. То же верно для операций над **`A`**, вызываемых **`B`**. Объект, вызывающий операцию, называется объект-клиент, а объект, который предоставляет операцию, - объект-сервер. Обычно отправитель сообщения знает получателя, но обратное необязательно.

В установившемся состоянии структуры классов и объектов системы должны быть согласованы. Если мы показываем на диаграмме операцию **`M`** на классе **`B`**, вызванную по связи **`L`**, то м должна быть объявлена в спецификации **`B`**, или в спецификациях его суперклассов.

Как показано на рис. 5-24, рядом с соответствующей связью на диаграмме можно записать набор сообщений. Каждое сообщение состоит из следующих трех элементов:

- **`D`** символ синхронизации, обозначающий направление вызова
- **`M`** вызов операции или извещение о событии
- **`S`** необязательный порядковый номер.

Мы показываем направление сообщения стрелкой, указывающей на объект-сервер. Этот символ означает простейшую форму передачи сообщений, семантика которой гарантирована только в присутствии единственного потока контроля. Существуют более развитые формы синхронизации, которые применимы в случае нескольких потоков. О них мы расскажем в следующем разделе.

Вызов операции - наиболее общая форма сообщения. Она подчиняется ранее описанному синтаксису операций, но, в отличие от него, здесь могут быть приведены фактические параметры, подходящие к сигнатуре операции:

- **`N()`** только имя операции
- **`RN(arguments)`** возвращаемое значение, имя и фактические параметры операции.

Сопоставление фактических параметров с формальными осуществляется в порядке следования. Если возвращаемый операцией объект или фактические параметры используют неквалифицированные имена, совпадающие с другими неквалифицированными именами на диаграмме, то подразумевается, что они именуют одинаковые объекты, а следовательно, их классы должны подходить к сигнатуре операции. Таким образом, мы можем представлять взаимодействия, в ходе которых объекты передаются в качестве параметров или возвращаются, как результат операции.

Сообщение может извещать о событии. Оно подчиняется определенному ранее синтаксису событий, и, следовательно, может представлять символьное имя, объект или имя некоторой операции. Во всяком случае, имя события должно быть определено на соответствующей классу объекта-сервера диаграмме переходов и состояний. Если извещение о событии является операцией, то оно может включать фактические параметры.

Если порядковый номер явно не указан, то сообщение может быть послано независимо от других сообщений, указанных на данной диаграмме объектов. Чтобы указать явный порядок событий, мы можем их пронумеровать. Нумерация начинается с единицы и добавляется как необязательный префикс к вызову операции или извещению о событии. Порядковый номер показывает относительный порядок посылки сообщений. Сообщения с одинаковыми номерами не упорядочены друг относительно друга; сообщение с меньшим порядковым номером посылается до сообщения с большим номером. Повторение порядковых номеров или их отсутствие говорит о частичной упорядоченности сообщений.

Пример. На рис. 5-25 показана диаграмма объектов для нашего тепличного хозяйства в контексте категории классов **Planning** (планирование; описана на рис. 5-7). Цель этой диаграммы - проиллюстрировать сценарий выполнения обычной функции системы, а именно, прогнозирование затрат на сбор урожая некоторого посева.

Выполнение этой функции требует сотрудничества нескольких различных объектов. Сценарий начинается с вызова объектом **PlanAnalyst** (анализатор планов) операции **timeToHarvest()** (время собирать урожай) над утилитой класса **PlanMetrics** (параметры планов). При этом объект с передается как фактический параметр операции. Затем утилита **PlanMetrics** вызывает операцию **status()** (состояние) на некотором неименованном объекте класса **GardeningPlan** (план выращивания). В пояснении говорится: "Надо проверить, что этот план действительно выполняется". В свою очередь, объект **GardeningPlan** вызывает операцию **maturationTime()** (время созревания) на выбранном объекте класса Grain**Crop** (посев зерновых), запрашивающую ожидаемое время созревания посева. Когда эта операция-селектор будет выполнена, управление возвращается объекту класса

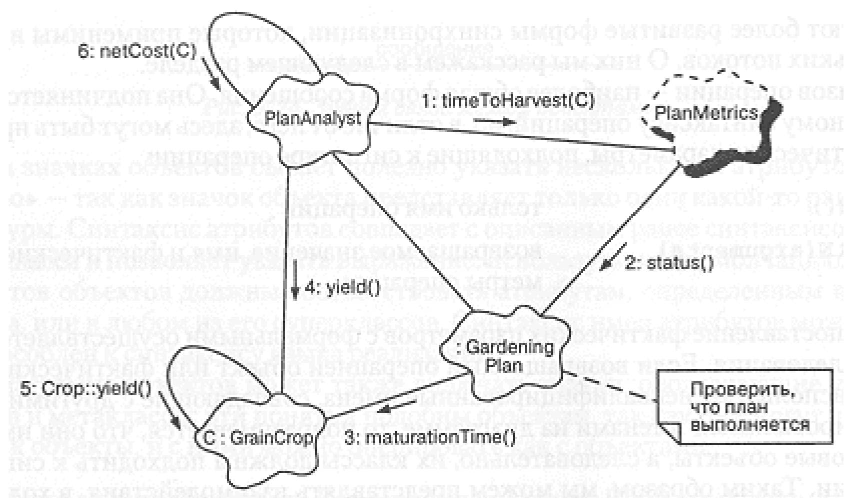


Рис. 5-25. Диаграмма объектов гидропонной системы

PlanAnalyst, который затем непосредственно вызывает операцию **C.yield()**, унаследованную от суперкласса (операция **Crop::yield()**). Управление снова возвращается объекту класса **PlanAnalyst**, который продолжает сценарий, выполняя над собой операцию **netCost ()**.

На диаграмме показана связь между объектами классов **PlanAnalyst** и **GardeningPlan**. Хотя сообщения между ними не посылаются, связь отражает существование семантической зависимости между этими объектами.

Дополнительные понятия

То, что мы описали, составляет существенные элементы диаграммы объектов. Однако многие запутанные вопросы разработки требуют некоторого расширения используемых обозначений. Мы предупреждали при описании диаграмм классов и хотим подчеркнуть опять: дополнительные понятия должны использоваться только при необходимости.

Роли, ключи и ограничения. Выше мы говорили, что на диаграмме классов при изображении ассоциации рядом с нею может быть написана ее роль, обозначающая намерение или мощность связи одного класса с другим. Для некоторых диаграмм объектов полезно заново написать эту роль при указании связи между объектами. Такая метка помогает объяснить, почему один объект оперирует над другим.

Рис. 5-26 дает пример использования этого дополнительного обозначения. Здесь мы видим, что некоторый объект класса **PlanAnalyst** заносит информацию об определенном посеве (Crop) в анонимный объект **CropEncyclopedia** (энциклопедия посевов) и делает это, пока находится в роли **Автор**.

Используя те же обозначения, что и для диаграммы классов, мы можем указать ключи или ограничения, ассоциированные с объектом или связью.

Поток данных. Как было описано в главе 3, данные могут передаваться по или против направления посылки сообщения. Иногда явное указание направления передачи данных помогает объяснить семантику конкретного сценария. Мы используем для этого значок, заимствованный из обозначений структурного проектиро-

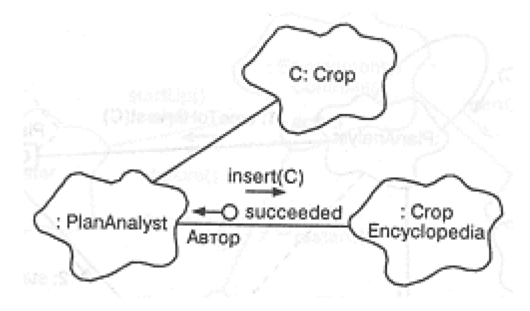


Рис. 5-26. Роли

вания. На рис. 5-26 дан пример его использования: здесь показано, что после завершения сообщения **insert** (вставить) возвращается значение **succeeded** (успех). Передаваться и возвращаться может либо объект, либо значение.

Видимость. В некоторых запутанных сценариях полезно указать точно, насколько один объект видит другие. Ассоциации на диаграммах классов обозначают семантическую зависимость между классами, но не указывают точно, насколько их экземпляры видят друг друга. С этой целью мы можем украсить связи на наших диаграммах значками, иллюстрирующими видимость одного объекта другим. Эта информация важна и для инструментальных программ, генерирующих код, или наоборот, восстанавливающих по коду логическую модель.

Рис. 5-27 уточняет рис. 5-25 и содержит несколько украшений, дающих информацию о видимости. Они похожи на украшения для

физического вхождения на диаграмме классов. Внутри этих украшений помещены буквенные обозначения типа видимости.

Например, канал связи от объекта **PlanAnalyst** к утилите классов **PlanMetrics** помечен буквой **G**; это значит, что утилита класса глобальна. Объект с по-разному виден объекту **PlanAnalyst** и объекту : **GardeningPlan**: с точки зрения первого объект с класса **GrainCrop** виден как параметр некоторой операции (обозначается буквой **P**); с точки зрения второго **C** виден как атрибут или поле, то есть как часть агрегированного объекта (обозначен буквой **F** (field)).

Вообще, для указания видимости могут быть использованы следующие обозначения:

- **G** сервер глобален для клиента
- **P** сервер является параметром некоторой операции клиента
- **F** сервер является частью клиента
- **L** сервер локально определен в области видимости клиента.

В соответствии с украшением для физического вхождения, украшение для видимости представляет собой незакрашенный квадратик с буквой (если объект используется совместно) или закрашенный квадратик с буквой (если он не используется совместно). Если украшение видимости не указано, это означает, что решение о точном типе видимости осталось не уточненным. На практике эти украшения прилагаются только к нескольким ключевым каналам связи на диаграмме объектов. Наиболее часто эти украшения указываются для отношения "часть/целое" (агрегация) между двумя объектами; второе наиболее общее их использование -

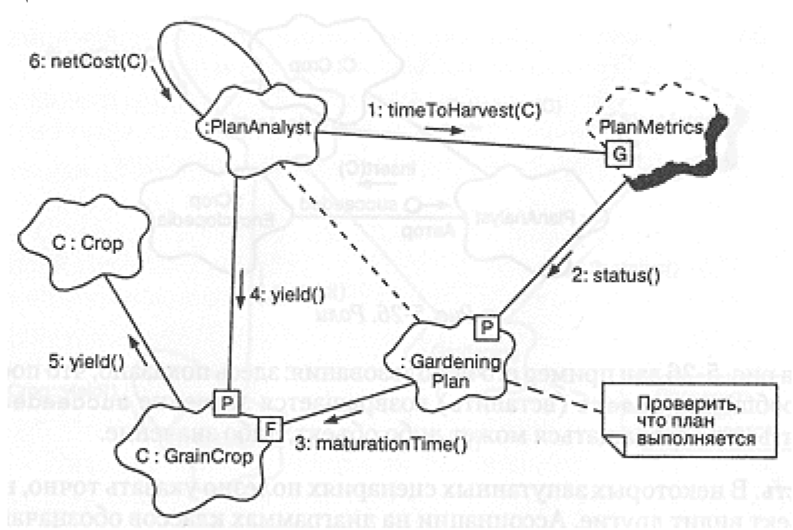


Рис. 5-27. Значки видимости

для представления объектов, которые по сценарию диаграммы посылаются как параметры.

Активные объекты и синхронизация. Как отмечалось в главе 3, некоторые объекты могут быть активными, то есть им отводится отдельный поток управления. Другие объекты могут существовать только в однопоточной среде. Третьи, будучи по природе однозадачными, гарантированно переносятся в многопоточную среду.

В каждом из этих случаев мы должны ответить на два вопроса: как выделить активные объекты, управляющие сценарием, и как представить различные формы синхронизации объектов.

Ранее, говоря о дополнительных элементах спецификаций класса, мы заметили, что есть четыре типа семантики: последовательная, защищенная, синхронизированная и активная. По существу, все объекты класса наследуют

соответствующую семантику класса; все объекты считаются последовательными, если явно не указано обратное. Мы можем явно показать многозадачную семантику объекта на диаграмме объектов, указав в левом нижнем углу значка объекта одно из слов **sequential**, **guarded**, **synchronous** или **active**. Например, на рис. 5-28 мы видим, что объекты **H**, **C** и некий экземпляр класса **EnvironmentalController** - активные. Немаркированные объекты, такие как **L**, считаются последовательными.

Символ синхронизации сообщений, введенный ранее (простая стрелка), представляет обычную последовательную передачу сообщения. Однако, при наличии нескольких потоков управления мы должны указывать и другие формы синхронизации. Пример на рис. 5-28, возможно, несколько надуманный, иллюстрирует различные типы синхронизации сообщений, которые могут появиться на диаграмме объектов. Сообщение **turnON()** (включить) - пример простой посылки сообщения; оно изображается простой стрелкой. Семантика простой посылки сообщения гарантирована только в однопоточной среде. Остальные сообщения из этого примера используют некоторые формы синхронизации процессов. Все такие дополнительные виды синхронизации применяются только к серверам, которые не являются последовательными.

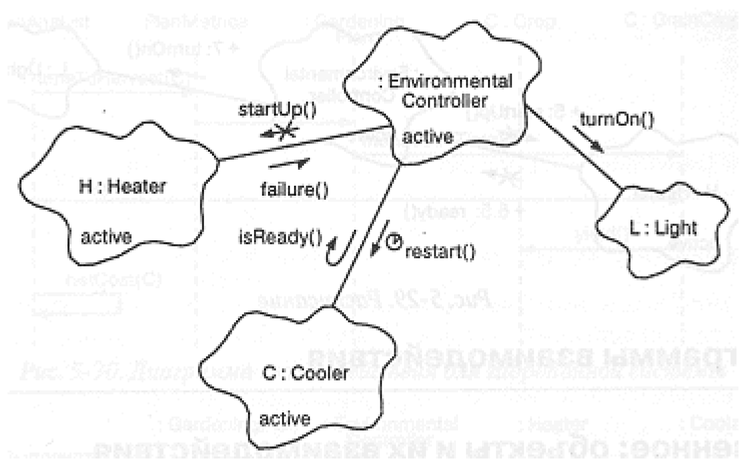


Рис. 5-28. Активные объекты и синхронизация

Например, сообщение **startUp()** - синхронизированное, то есть клиент будет ждать до тех пор, пока сервер не примет сообщение. Посылка синхронизированного сообщения эквивалентна механизму свиданий задач в языке Ada (rendezvous). В случае сообщения **isReady()** клиент отложит сообщение, если сервер н сможет его немедленно обработать. Сообщение **restart()** будет отложено клиентом, если сервер не может его обработать за указанный промежуток времени.

В каждом из трех последних случаев клиент должен ждать, пока сервер обработает сообщение, или отложить пересылку, после чего может быть возобновлено управление. Сообщение **failure** имеет другую семантику. Это пример несинхронизированного сообщения, которое подразумевает, что клиент посылает событие серверу для обработки, сервер ставит сообщение в очередь, а клиент продолжает работать. Такие асинхронные сообщения сродни прерываниям.

Расписание. В программах, имеющих ограничения по времени, важно отслеживать чистое время с момента начала каждого сценария. Для обозначения относи тельного времени (в секундах) мы ставим знак плюс. Например, на рис. 5-29 сообщение **startUp()** вызывается в первый раз спустя 5 секунд после начала сценария далее, через 6.5 секунд после начала сценария следует сообщение **ready()** и затем, спустя 7 секунд после начала сценария, - сообщение **turnOn()**.

Спецификации

Как и для диаграмм классов, за каждым элементом диаграммы объектов могут стоять спецификации. Спецификации объектов и их связей не несут никакой иной информации, кроме уже описанной. С другой стороны, спецификации диаграмм объектов как целого могут сообщить кое-что важное. Как упоминалось ранее, каждая диаграмма объектов существует в контексте. В спецификации контекст указывается следующим образом:

```
Context: глобальный | категория | класс | операция
```

В частности, область видимости диаграммы объектов может быть глобальной, или в контексте указанной категории классов, класса или операции (включая, как методы, так и свободные подпрограммы).

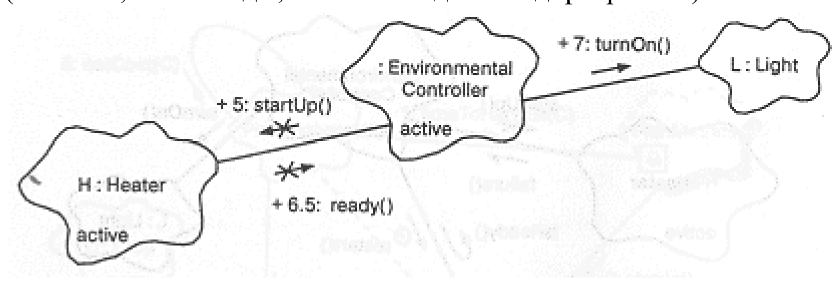


Рис. 5-29. Расписание

5.5. Диаграммы взаимодействия

Существенное: объекты и их взаимодействия

Диаграмма взаимодействии используется, чтобы проследить выполнение сценария в том же контексте, что и диаграмма объектов.[15] В известной степени диаграмма взаимодействия есть просто другой способ представления диаграммы объектов. Например, на рис. 5-30 мы видим диаграмму взаимодействия, которая дублирует большую часть семантики диаграммы объектов, показанной на рис. 5-25. Преимущество диаграммы взаимодействий в том, что на ней легче читается порядок посылки сообщений, а преимущество диаграммы объектов в том, что она лучше подходит для многих объектов со сложными вызовами и допускает включение другой информации: связи, значения атрибутов, роли, блок-схемы и видимость. Так как оба типа диаграмм имеют неоспоримые достоинства, мы пользуемся в нашем методе обоими.[16]

Диаграммы взаимодействия не вводят новых понятий или обозначений. Скорее, они берут существенные элементы диаграммы объектов и перестраивают их. Как показывает рис. 5-30, диаграмма взаимодействий внешне напоминает таблицу. Имена объектов диаграммы взаимодействий (те же, что и на диаграмме объектов) записываются горизонтально в верхней ее строке. Под каждым из них рисуется вертикальная пунктирная линия. Отправления сообщений (которые могут обозначать события или вызовы операций) показываются горизонтальными стрелками, с тем же синтаксисом и обозначениями синхронизации, что и на диаграмме объектов. Линия, обозначающая посылку сообщения, проводится от вертикали клиента к вертикали сервера. Первое сообщение показывается на самом высоком уровне,

---

[15] Эти диаграммы обобщают диаграммы трассировки событий Румбаха и диаграммы взаимодействий Джекобсона [15].

[16] Диаграммы объектов и диаграммы взаимодействий настолько близки по семантике, что инструментальные средства могут генерировать одну диаграмму из другой с минимальной потерей информации.

второе ниже и т. д., таким образом отпадает надобность в их порядковых номерах.

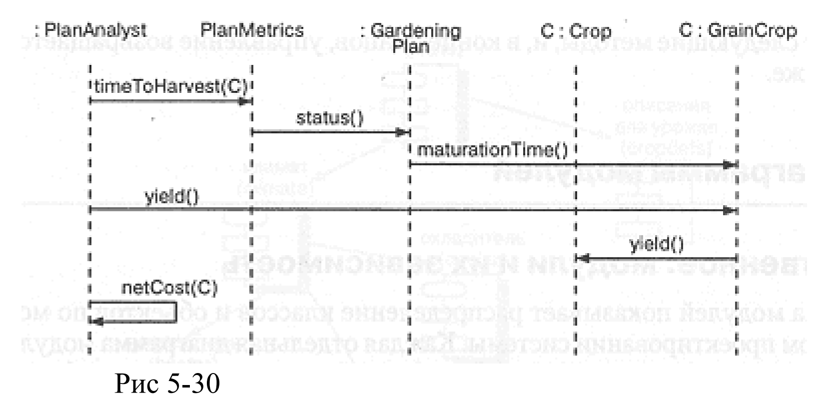


Рис 5-30

Диаграммы взаимодействий часто лучше диаграмм объектов передают семантику сценариев на ранних этапах жизненного цикла разработки, когда еще не идентифицированы протоколы отдельных классов. Как мы расскажем в следующей главе, в начале разработки диаграммы взаимодействий обычно сконцентрированы скорее на событиях, чем на операциях, потому что события помогают определить

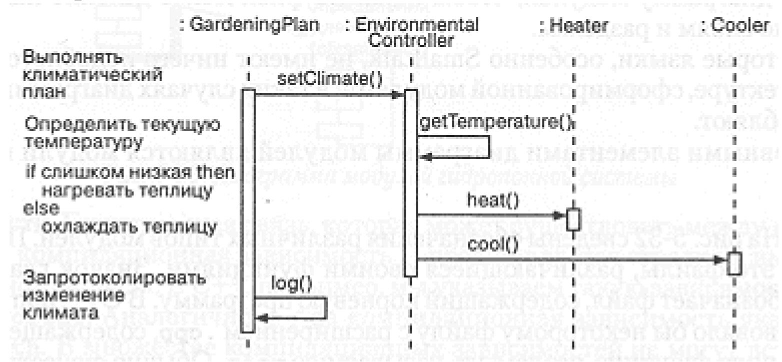


Рис. 5-31. Пояснения и переход управления

границы системы. Когда же уточнились структуры классов, акцент смещается к диаграммам объектов, семантика которых более выразительна.

Дополнительные понятия

Диаграммы взаимодействия концептуально очень просты, но есть два поясняющих элемента, которые позволяют сделать их более выразительными при наличии сложных шаблонов взаимодействия.

Пояснения. Для сложных сценариев, использующих условия или итерации, диаграмма объектов может быть дополнена пояснениями. Как показано на примере (рис. 5-31), пояснения могут быть подписаны к любому сообщению слева от диаграммы на соответствующем уровне простым текстом, с элементами структуризации, или с использованием синтаксиса языка реализации.

Передача управления. Ни простейшие диаграммы объектов, ни диаграммы взаимодействий не показывают передач управления. Например, если мы показали, что объект **`A`** посылает сообщения **`X`** и **`Y`** другим объектам, то остается неясным, являются ли сообщения **`X`** и **`Y`** независимыми сообщениями из **`A`** или они были вызваны как части некоторого объемлющего сообщения **`Z`**. Как показано на рис. 5-31, мы можем нарисовать на вертикальной линии каждого объекта полоски, показывающие периоды, когда управление находится в этом объекте. На этом примере мы видим, что всем

руководит анонимный экземпляр класса **GardeningPlan**, который, выполняя климатический план, вызывает другие методы, которые, в свою очередь,вызывают следующие методы, и, в конце концов, управление возвращается обратно к нему же.

5.6. Диаграммы модулей

Существенное: модули и их зависимость

Диаграмма модулей показывает распределение классов и объектов по модулям в физическом проектировании системы. Каждая отдельная диаграмма модулей представляет некоторый ракурс структуры модулей системы. При разработке мы используем диаграмму модулей, чтобы показать физическое деление нашей архитектуры по слоям и разделам.

Некоторые языки, особенно Smalltalk, не имеют ничего подобного физической архитектуре, сформированной модулями; в таких случаях диаграммы модулей не употребляют.

Основными элементами диаграммы модулей являются модули и их зависимости.

Модули. На рис. 5-32 сведены обозначения различных типов модулей. Первые три значка - это файлы, различающиеся своими функциями. Значок главной программы обозначает файл, содержащий корневую программу. В C++, например, это соответствовало бы некоторому файлу с расширением **.cpp**, содержащему привилегированную функцию-неэлемент, называемую **main**. Обычно существует ровно один такой модуль на программу. Значок описания и значок тела обозначают файлы, которые содержат, соответственно, описания и реализации. В C++, например, модуль описаний соответствует заголовочному файлу с расширением **.h**, а модуль тела - файлу с текстом программы с расширением **.cpp**.

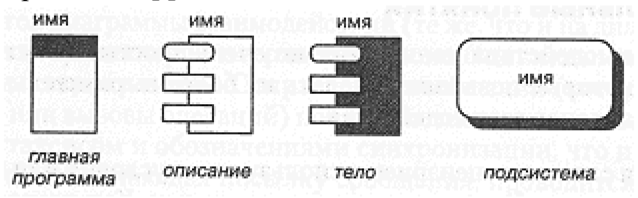


Рис. 5-32. Значки модулей и подсистем

Смысл значка подсистемы мы раскроем в следующем разделе. Каждый модуль должен иметь имя; обычно это имя соответствующего физического файла в каталоге проекта. Как правило, такие имена пишутся без суффиксов, которые опознаются по типу значка. Если имя чересчур длинно, мы, как обычно, либо сокращаем его, либо расширяем значок. Каждое полное имя файла должно быть уникально в содержащей его подсистеме. В соответствии с правилами конкретной среды разработки, мы можем наложить ограничения на имена, такие, как условие на префиксы или требование уникальности в системе.

Каждый модуль содержит либо описание, либо определение классов и объектов, а также другие конструкции языка. По идее, "раскрыв" значок модуля на диаграмме, мы должны попасть внутрь соответствующего файла.

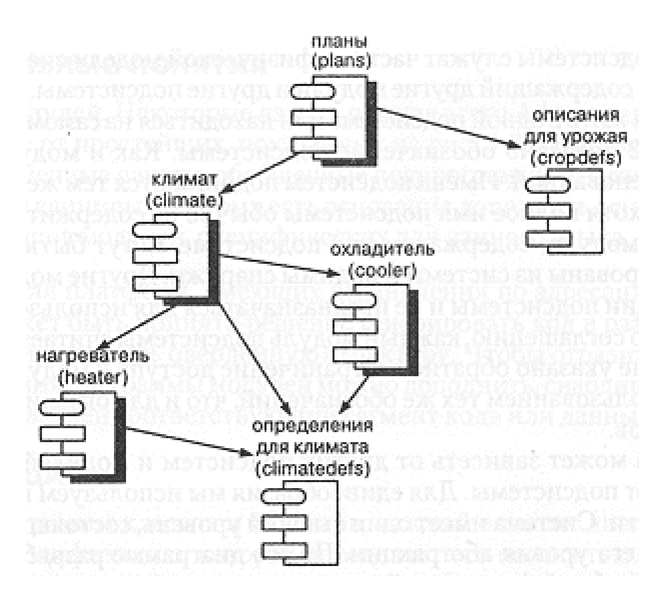


Рис. 5-33. Диаграмма модулей гидропонной системы

Зависимости. Единственная связь, которая может существовать между двумя модулями, - компиляционная зависимость - представляется стрелкой, выходящей из зависимого модуля. В C++, например, мы указываем такую зависимость директивой **`#include`**. Аналогично, в Ada компиляционная зависимость указывается фразой **`with`**. В множестве компиляционных зависимостей не могут встречаться циклы. Чтобы определить частичную упорядоченность компиляций, достаточно выполнить частичное упорядочение структуры модулей системы.

Пример. На рис. 5-33 показан пример обозначений модулей, в архитектуре системы тепличного гидропонного хозяйства. Мы видим здесь шесть модулей. Два из них, **`climatedefs`** и **`cropdefs`**, являются только описаниями и служат для предоставления общих типов и констант. Остальные четыре включают в себя и тела, и описания: это типичный стиль построения диаграмм модулей, так как описания и тела очень тесно связаны. На рисунке эти две части совмещены, и зависимость тела от описания получилась скрытой, хотя реально она существует. Также оказалось скрытым имя тела, но, по нашему соглашению, имена тела и описания различаются лишь суффиксами (**`.cpp`** и **`.h`**).

Зависимости на этой диаграмме предполагают частичное упорядочение компиляции. Например, тело модуля **`climate`** зависит от описания **`heater`**, которое, в свою очередь, зависит от описания **`climatedefs`**.

Существенное: подсистемы

Как объяснялось в главе 2, большие системы могут быть разложены на несколько сотен, если не тысяч, модулей. Пытаться разобраться в физической архитектуре такой системы без ее дополнительного структурирования почти безнадежно. На практике разработчики стремятся следовать неформальному соглашению собирать связанные между собой модули в структуры типа каталогов. По этим соображениям мы введем понятие подсистемы на диаграмме модулей. Подсистемы представляют собой совокупности логически связанных модулей, примерно как категория классов представляет совокупность классов.

Подсистемы. Подсистемы служат частями физической модели системы. Подсистема - это агрегат, содержащий другие модули и другие подсистемы. Каждый модуль в системе должен жить в одной подсистеме или находиться на самом верхнем уровне.

На рис. 5-32 показано обозначение подсистемы. Как и модуль, подсистема должна быть именованной. Имена подсистем подчиняются тем же правилам, что и имена модулей, хотя полное имя подсистемы обычно не содержит суффиксов.

Некоторые модули, содержащиеся в подсистеме, могут быть общедоступны, то есть экспортированы из системы и видимы снаружи. Другие модули могут быть частью реализации подсистемы и не предназначаться для использования внешними модулями. По соглашению, каждый модуль подсистемы считается общедоступным, если явно не указано обратное. Ограничение доступа к модулям реализации достигается использованием тех же обозначений, что и для ограничения доступа в категории классов.

Подсистема может зависеть от других подсистем и модулей; модуль может также зависеть от подсистемы. Для единообразия мы используем прежнее обозначение зависимости. Система имеет один высший уровень, состоящий из подсистем и модулей высшего уровня абстракции. По его диаграмме разработчик получает представление об общей физической архитектуре системы.

Пример. На рис. 5-34 показан высший уровень диаграммы модулей для нашей системы тепличного хозяйства. Раскрыв любую из показанных семи подсистем, мы обнаружим все ее модули.

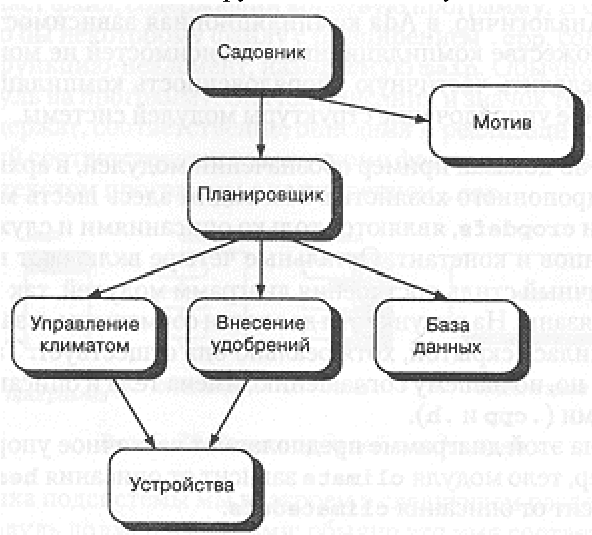


Рис. 5-34. Диаграмма модулей верхнего уровня для гидропонной системы

Рассмотрим, как связаны физическая и логическая (рис. 5-7) архитектуры этой системы. Они практически изоморфны, хотя имеются небольшие различия. В частности, мы приняли решение отделить классы устройств нижнего уровня от категорий классов `Климат` и `Удобрения`, и поместить соответствующие им модули в одну подсистему, названную `Устройства`. Кроме того, мы разделили категорию классов `Теплица` на две подсистемы, названные `УправлениеКлиматом` и `ВнесениеУдобрений`.

Дополнительные понятия

Другие типы модулей. Некоторые языки, прежде всего Ada, определяют типы модулей, отличные от простейших, показанных на рис. 5-32. Например, Ada предусматривает обобщенные пакеты, обобщенные подпрограммы и задачи как раздельно компилируемые единицы. Поэтому есть основания дополнить основные обозначения значками типов модулей, специфических для данного языка.

Сегментация. Для платформ, имеющих ограничения по адресации или физической памяти, может быть принято решение генерировать код в различных сегментах, или даже организовать оверлейную структуру. Чтобы отразить такую сегментацию обозначения диаграммы модулей можно дополнить, снабдив каждый модуль меткой, обозначающей соответствующий сегмент кода или данных.

Спецификации

Так же, как диаграммы классов и объектов, каждый элемент диаграммы модулей может иметь спецификацию, которая определяет его полностью. Спецификации модулей и их зависимостей содержат только ту информацию, которая уже описана в этом разделе, поэтому мы не будем их рассматривать.

В интегрированной инструментальной среде, поддерживающей наши обозначения, разумно использовать диаграммы модулей для визуализации программных модулей системы. "Раскрытие" модуля или подсистемы на диаграмме модулей открывает соответствующий физический файл или каталог и наоборот.

5.7. Диаграммы процессов

Существенное: процессоры, устройства и соединения

Диаграммы процессов используются, чтобы показать распределение процессов по процессорам в физическом проекте системы. Отдельная диаграмма процессов показывает один ракурс структуры процессов системы. При разработке проекта мы используем диаграмму процессов, чтобы показать физическую совокупность процессоров и устройств, обеспечивающих работу системы.

Основные элементы диаграммы процессов - процессоры, устройства и их соединения.

Процессоры. На рис. 5-35 показано обозначение процессора. Процессор - часть аппаратуры, способная выполнять программы. Каждый процессор должен иметь имя; никаких особых ограничений на имена процессоров нет, так как они обозначают "железо", а не программы.

Мы можем дополнить значок процессора списком процессов. Каждый процесс в таком списке обозначает или главную программу (функцию main из диаграммы модулей) или имя активного объекта (из диаграммы объектов).

Устройства. На рис. 5-35 показано обозначение устройства. Устройство - это часть аппаратной платформы, не способная выполнять программы (по крайней мере, в

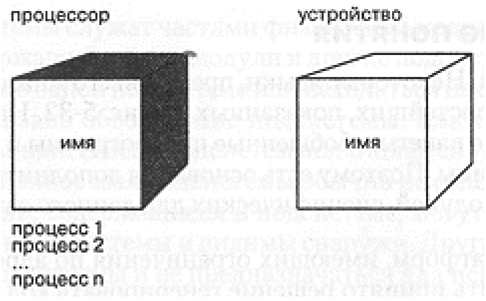


Рис. 5-35. Значки процессора и устройства

нашей логической модели). Как и процессорам, устройствам требуются имена, на которые не накладывается никаких существенных ограничений.

Соединения. Процессоры и устройства должны сообщаться друг с другом. На диаграмме процессов мы изображаем соединения между ними ненаправлеными линиями. Соединение обычно представляет

непосредственную связь между аппаратурой, например, RS232, Ethernet, или даже доступ к разделяемой памяти, но эта связь может быть и не прямой, например, "Земля-спутник". Соединения обычно считаются двунаправленными, но при необходимости к их обозначению можно добавить стрелку, чтобы указать направление. Любое соединение может иметь необязательную именующую его метку.

Пример. На рис. 5-36 представлен пример использования этих обозначений, описывающий физическую архитектуру тепличной системы. Мы видим, что разработчики решили использовать четыре компьютера, один в качестве рабочего места оператора и по одному на каждую теплицу. Процессы, запущенные на выделенных теплицам компьютерах, не могут сообщаться друг с другом непосредственно, а только через рабочую станцию. Для простоты мы решили не показывать на этой диаграмме никаких устройств, хотя предполагаем, что система содержит большое число исполнительных устройств и датчиков.

Дополнительные понятия

Обозначения. На рис. 5-35 показаны стандартные обозначения, которые мы используем для процессора и устройства, но разумно и даже желательно учесть возможность их изменения. Например, можно было бы ввести специальные значки для встроенного микрокомпьютера (процессор), диска, терминала и выпрямителя тока (устройства), и использовать их на диаграммах процессов вместо стандартных. Поступая таким образом, мы предлагаем визуализацию физической платформы нашей реализации, которая предназначена непосредственно техникам и системщикам, а также конечным пользователям системы, которые, вероятно, не являются специалистами в разработке программного обеспечения.

Вложенность. Физическая конфигурация системы бывает очень сложной и может представлять собой иерархию процессоров и устройств. В таких случаях полезно иметь возможность выделить группы процессоров, устройств и соединений, так же, как категории классов представляют логическое группирование классов и объектов. Мы изображаем такие группы именованными пунктирными прямоугольника-

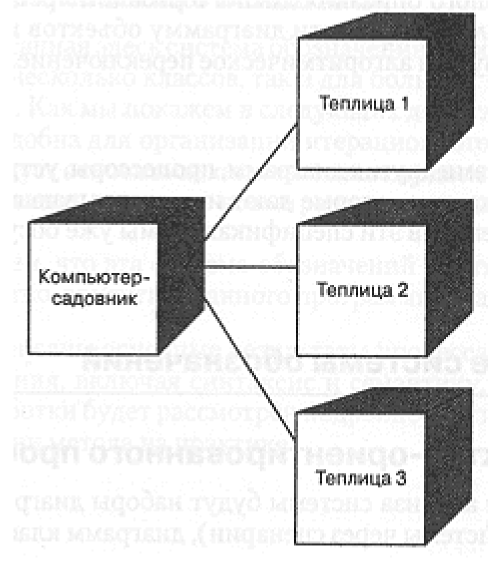


Рис. 5-36. Диаграмма процессов гидропонной системы

ми с закругленными углами. Мы можем раскрыть такой значок на диаграмме процессов и обнаружить вложенные процессоры и устройства. Не представляет затруднений определить соединения между этими группами.

Процесс разработки организован как придется и нередко хаотичен. На этой стадии

налаживание элементарного управления проектом - уже прогресс.Планирование процессов. Мы должны некоторым образом определить порядок выполнения процессов на каждом процессоре. Имеется пять основных способов планирования, и мы можем указать на диаграмме для каждого процессора, какой из них использован, добавив к его значку одно из следующих имен:

Для более подробного описания диспетчеризации процессов на конкретном процессоре бывает полезно привести диаграмму объектов или взаимодействий, особенно если используется алгоритмическое переключение.

Спецификации

По аналогии с элементами других диаграмм, процессоры, устройства и соединения могут иметь спецификации, которые дают их исчерпывающее определение. Всю информацию, включаемую в эти спецификации, мы уже обсудили в текущем разделе.

5.8. Применение системы обозначений

Результат объектно-ориентированного проектирования

Обычно результатами анализа системы будут наборы диаграмм объектов (чтобы выразить поведение системы через сценарии), диаграмм классов (чтобы выразить роли и обязанности агентов по поддержанию заданного поведения системы) и диаграммы состояний и переходов (чтобы показать упорядоченное событиями поведение этих агентов). Проектирование системы, в которое входит разработка ее архитектуры и реализации, порождает диаграммы классов, объектов, модулей, процессов, а также динамические ракурсы этих диаграмм.

Существует сквозная связь между этими диаграммами, позволяющая нам проследить требования от реализации обратно к спецификации. Начав с диаграмм процессов, можно найти главную программу, которая определена на некоторой диаграмме модулей. Эта диаграмма модулей содержит наборы классов и объектов, определения которых мы найдем на подходящих диаграммах классов или объектов. Наконец, определения отдельных классов указывают на наши исходные требования, потому что эти классы, в общем, непосредственно отражают словарь предметной области.

Описанной в этой главе системой обозначений можно пользоваться вручную, хотя, конечно, она просто напрашивается на автоматизацию. Автоматизированным инструментам проектирования можно поручить проверку целостности, ограничений и полноты документации. Они также помогают разработчику легко и быстро просматривать результаты анализа и разработки. Например, глядя на диаграмму модулей, разработчик может пожелать выяснить устройство конкретного механизма, и автоматизированный инструмент поможет ему отыскать все классы, объявленные в каком-то модуле. А от диаграммы объектов, описывающей сценарий, в котором использован один из классов, разработчик может перейти к структуре наследования этого класса. Наконец, если в сценарии есть активный объект, разработчик может использовать автоматизированный инструмент проектирования, чтобы отыскать процессор, которому выделен соответствующий поток управления, и увидеть анимированное поведение конечного автомата класса на этом процессоре. Использование автоматизированных инструментов позволяет освободить разработчика от бремени согласования деталей, позволяя ему сосредоточиться на творческих аспектах процесса проектирования.

Увеличение и уменьшение масштаба

Мы считаем, что описанная здесь система обозначений годится как для маленьких систем, содержащих несколько классов, так и для больших проектов с несколькими тысячами классов. Как мы покажем в следующих двух главах, эта система обозначений особенно удобна для организации итерационного процесса разработки. К диаграммам не следует относиться как к застывшему догмату, а скорее наоборот, нужно постоянно отражать на них все новые решения, принятые в процессе проектирования.

Мы также считаем, что эта система обозначений годится для реализации на разных языках объектно-ориентированного программирования.

В этой главе были описаны основные результаты процесса объектно-ориентированного проектирования, включая синтаксис и семантику. В двух последующих главах процесс разработки будет рассмотрен подробнее. Оставшиеся пять глав повествуют о применении метода на практике.

Выводы

Проектирование - это не рисование диаграмм; диаграммы просто отражают результаты проектирования.

При проектировании сложной системы важно рассмотреть ее в различных ракурсах - как с точки зрения логической/физической структуры, так и статической/динамической семантики.

Система обозначений объектно-ориентированного проектирования включает четыре основных диаграммы (классов, объектов, модулей, процессов) и две дополнительные (состояний и переходов, взаимодействий).

Диаграмма классов показывает, какие существуют классы и связи между ними в логической структуре системы. Конкретная диаграмма классов - один из ракурсов полной структуры классов системы.

Диаграмма объектов показывает, какие существуют объекты и связи между ними в логической структуре системы. Диаграмма объектов используется для представления сценария.

Диаграмма модулей показывает распределение классов и объектов по модулям в физической структуре системы. Диаграмма модулей - один из ракурсов модульной архитектуры системы.

Диаграмма процессов показывает распределение процессов по процессорам в физической структуре системы. Каждая диаграмма процессов - один из ракурсов архитектуры процессов системы.

Диаграмма состояний и переходов показывает: (1) пространство состояний экземпляров данного класса; (2) события, которые влекут переход из одного состояния в другое; (3) действия, которые происходят при изменении состояния.

Диаграмма взаимодействий позволяет следить за выполнением сценария в контексте диаграммы объектов.

Дополнительная литература

Со времени выхода первого издания этой книги я без устали старался ввести в метод Буча лучшие элементы обозначении, принадлежащие другим методологам, особенно Румбаху (Rumbaugh) и Джекобсону (Jacobson), удалял и упрощал элементы, неудачные или имеющие сомнительную пользу. В то же время, концептуальное единство системы обозначений береглось как зеница ока. Данная глава - кульминация этих усилий.

Об обозначениях в разработке программного обеспечения написано чрезвычайно много; книга Мартина и МакКлюра (Martin and McClure) [H 1988] служит хорошим общим справочником по многим традиционным подходам. Грэхам (Graham) [F 1991] дал обзор ряда нотаций, специфичных для объектно-ориентированных методов.

Ранняя форма описанной в этой главе системы обозначений была впервые документирована в работе Буча (Booch) [F 1981]. Эта система в дальнейшем развилась и включила выразительные средства: семантических сетей (Стилингс и др. (Stillings et al.) [A 1987] и Барри Фейгенбаум (Barrand Feigenbaum) [J 1981]),диаграммы "сущность-отношение" (Чэн (Chen) [E 1976]), модели сущности (Росс (Ross) [F 1987]), сети Петри (Petri) (Пе-терсон (Peterson) [J 1977], Сахару (Sahraoui) [F 1987] и Бруон и Балзамо (Bruon and Balsamo) [F 1986]), ассоциации (Румбах (Rumbaugh) [F 1991]) и карты состояний (Харел (Harel) [F 1987]). Особенно интересна работа Румбаха, поскольку, как он заметил, в наших подходах больше сходства, чем различий.

Значки для объектов и классов были инспирированы iAPX 432 [D 1981]. За основу изображения для объектных диаграмм были взяты обозначения Сейдвица (Seidewitz) [F 1985]. Для семантики параллельности были приспособлены обозначения Бура (Buhr) [F1988,1989].

Чэн (Chang) [G 1990] дал хороший обзор более общих аспектов визуальных языков.

Глава 6

Процесс

Программисты-любители все время ищут какой-то волшебный инструмент, который мог бы сделать процесс разработки программ тривиальным. Признак профессионализма - понимание того, что такой панацеи не существует. Любители стремятся действовать по "поваренной книге"; профессионалы же знают, что безупречно грамотный подход ведет к нелепым проектным решениям. За словом "система проектирования" разработчики пытаются спрятаться от ответственности за ошибки в проектных решениях. Любители либо игнорируют документацию вообще, либо выстраивают весь проект вокруг нее, заботясь больше о том, как продукт выглядит на бумаге, чем о его сути. Профессионал признает, что без документации не обойтись, но никогда не поступится ради нее полезными архитектурными новациями.
Процесс объектно-ориентированного анализа и проектирования не сводится к сумме рецептов, однако он определен достаточно хорошо, чтобы быть предсказуемым и воспроизводимым в умелых руках. В этой главе мы подробно рассмотрим его как итеративно развивающийся процесс, описав цели, виды деятельности, результаты и меры прогресса, характерные для его различных фаз.

6.1. Основные принципы

Характерные черты удачных проектов

Удачным проектом мы назовем тот, который удовлетворил (по возможности, превзошел) ожидания заказчика, уложился во временные и финансовые рамки, легко поддается изменению и адаптации. Пользуясь этим критерием, рассмотрим следующие две черты, которые оказались общими для всех встречавшихся нам удачных проектов, и, что замечательно, отсутствовали у тех, которые кажутся нам неудачными:

- Ясное представление об архитектуре создаваемой системы;
- Хорошо организованный итеративно развивающийся процесс работы над проектом.
-
- **Архитектура.** Признак добротности архитектуры - ее концептуальное единство и целостность. По утверждению Брукса, "концептуальная целостность в проектировании важнее всего" [1]. Как показано в главах 1 и 5, архитектура объектно-ориентированной программной системы содержит структуры классов и объектов, имеющие горизонтальное и вертикальное слоение. Обычно конечному пользователю нет дела до архитектуры системы. Однако, как указывает Страуструп, "ясная внутренняя структура" играет важную роль в построении системы, которая будет понятна, тестируема, устойчива и сможет развиваться и перестраиваться [2]. Более того, именно ясность архитектуры дает возможность выявить общие абстракции и механизмы, которые можно свести воедино, тем самым делая систему проще, меньше и надежнее.
- Не существует единственно верного способа классифицировать абстракции и разрабатывать архитектуру. В любой предметной области всегда достаточно глупейших путей проектирования, но, если поискать, можно найти и весьма элегантные. Как же отличить хорошую архитектуру от плохой?
- Как правило, хорошая архитектура тяготеет к объектной ориентированности. Это не означает, что любая объектно-ориентированная архитектура оказывается хорошей, или что хороша только объектно-ориентированная архитектура. Однако, как было показано в главах 1 и 2, применение принципов объектно-ориентированной декомпозиции приводит к архитектуре, обладающей требуемыми свойствами организованной сложности.
- Хорошей архитектуре присущи следующие свойства:
- Она представляет собой многоуровневую систему абстракций. На каждом уровне абстракции сотрудничают друг с другом, имеют четкий интерфейс с внешним миром и основываются на столь же хорошо продуманных средствах нижнего уровня.
- На каждом уровне интерфейс абстракции строго отграничен от реализации. Реализацию можно изменять, не затрагивая при этом интерфейс. Изменяясь

внутренне, абстракции продолжают соответствовать ожиданиям внешних клиентов.

- Архитектура проста, то есть не содержит ничего лишнего: общее поведение достигается общими абстракциями и механизмами.
- Мы различаем стратегические и тактические решения. *Стратегическое решение* имеет важное архитектурное значение и связано с высоким уровнем системы. Механизмы обнаружения и обработки ошибок, парадигмы интерфейса пользователя, политика управления памятью, устойчивость объектов, синхронизация процессов, работающих в реальном масштабе времени, - все это стратегические архитектурные решения. В противоположность этому, *тактическое решение* имеет только локальное архитектурное значение и поэтому обычно связано с деталями интерфейса и реализации абстракций. Протокол класса, сигнатура метода, выбор алгоритма - все это тактические архитектурные решения.
- Хорошая архитектура всегда демонстрирует баланс между стратегическими и тактическими решениями. При слабой стратегии даже очень изящно задуманный класс не сможет вполне соответствовать своей роли. Самые прозорливые стратегические решения будут разрушены, если не уделить должного внимания разработке отдельных классов. В обоих случаях пренебрежение архитектурой рождает программные эквиваленты анархии и неразберихи.
-
- **Цикл итеративного развития.** Рассмотрим две крайности - полное отсутствие формализованного жизненного цикла разработки и очень жесткие, строго соблюдаемые правила разработки. В первом случае мы имеем анархию; тяжким трудом (преимущественно нескольких своих членов) команда разработчиков в конце концов может родить что-то стоящее, но состояние проекта всегда будет неизмеримо и непредсказуемо. Следует ожидать, что команда отработает весьма неэффективно, а, может быть, и вообще не создаст ничего пригодного для передачи заказчику. Это - пример проекта в свободном падении.[17] Во втором случае мы имеем диктатуру, в которой инициативы наказуемы, экспериментирование, которое могло бы привнести больше элегантности в архитектуру, не поощряется, и действительные требования заказчика никогда корректно не доходят до разработчиков нижнего уровня, скрытых за настоящей бумажной стеной, воздвигнутой бюрократией.
- Встречавшиеся нам удачные объектно-ориентированные проекты не следовали ни анархическому, ни драконовскому жизненному циклу. Зато мы заметили, что удачная объектно-ориентированная архитектура создается в итеративно развивающемся процессе. Проектирование является *итеративным*, повторяющимся, в том смысле, что уже созданная архитектура вновь и вновь подвергается анализу и проектированию. При этом в каждом цикле анализ-проектирование-эволюция стратегические и тактические решения *развиваются*, приближаясь к требованиям конечного пользователя (часто даже не высказанным), оставаясь при этом простыми, надежными и открытыми для дальнейшего изменения.
- Итеративно развивающийся процесс является антитезой традиционного "водопада" и не сводится к одностороннему движению сверху-вниз или снизу-вверх. Обнадеживающие прецеденты этого стиля есть в опыте создания как аппаратуры, так и программ [3, 4]. Например, пусть надо сформировать штат фирмы, занимающейся проектированием и изготовлением сложной уникальной аппаратуры. Можно использовать горизонтальный подход, когда проект катится водопадом, так, что архитекторы передают его конструкторам, а те электронщикам. Это - пример проектирования сверху-вниз, когда мы приглашаем узких (хотя и глубоких) специалистов в своей области [5]. Можно пойти по другому пути, наняв мастеров на все руки, каждому из которых можно поручить вертикальный сегмент проекта от начала до конца. Это уже гораздо больше похоже на итеративно развивающийся процесс.

---

[17] Есть шанс, что проект в свободном падении приземлится благополучно, но вам не нужно ставить в связи с этим на кон будущее своей компании.

- По нашему мнению, процесс объектно-ориентированного проектирования не сводится к одностороннему движению сверху-вниз или снизу-вверх. Друк считает, что хорошо структурированные сложные системы можно создать методом "возвратного проектирования" (round-trip gestalt design). В этом методе основное внимание уделяется процессу поступательного итеративного развития путем совершенствования различных, но, тем не менее, совместимых между собой логических и физических моделей системы. Мы считаем, что возвратное проектирование составляет необходимую основу процесса объектно-ориентированного проектирования.
- В отдельных случаях решаемая задача может быть уже хорошо изучена и много раз запрограммирована. Процесс разработки можно привести в идеальный порядок: проектировщики новой системы уже понимают, какие абстракции являются главными; они уже знают, какие механизмы нужно использовать и каким, в общих чертах, будет поведение системы. Творчество все еще важно в таком процессе, но здесь проблема достаточно сужена и большинство стратегических решений предопределены. Тогда, поскольку риск исключен, можно достичь очень высоких показателей производительности [б]. Чем больше мы знаем о задаче, тем легче ее решить.
- Большинство промышленных задач не таковы: они связаны с балансированием уникальных требований к функциональности и эффективности и требуют полной творческой отдачи всего коллектива разработчиков. Более того, любая человеческая деятельность, которая требует творчества и инноваций, идет путем проб и ошибок, итеративно развивающегося процесса, который опирается на опыт, компетентность и талант каждого члена коллектива.[18] Так что нет и не будет стандартных рецептов для проектирования программных систем.

•Рациональный процесс проектирования

- Однако мы не можем обойтись без рецептов, описывая обещанную выше зрелую, воспроизводимую в любой организации технологию разработки. Поэтому мы и характеризовали ее, как управляемый итеративно развивающийся процесс - управляемый в том смысле, что он поддается проверке и измерению, но оставляет достаточную свободу для творчества.
- Упорядоченный процесс проектирования чрезвычайно важен для организаций, разрабатывающих программное обеспечение. Хэмфри перечисляет следующие пять уровней зрелости таких процессов [7]:
-

•• Начальный

Пользователь извне системы управляет переключением процессов.

• Ручное

Переключением процессов управляет некоторый алгоритм.

• Алгоритмическое

Процессам по очереди выделяется равное количество процессорного времени, обычно называемое квантом времени, по истечении которого управление передается другому процессу. Процесс может получать время в квантах и подквантах.

• Циклическое

---

[18] Эксперименты Кертиса и его коллег подкрепляют эти наблюдения. Они изучали работу профессиональных разработчиков программного обеспечения, записывая видеокамерой их действия и затем анализируя их содержание (анализ, проектирование, реализация и т. п.) и время на выполнение. В результате исследований был сделан вывод, что "создание программ представляется набором итеративных, плохо упорядоченных и взаимно перекрывающихся процессов под приспосабливающимся управлением... Развитие по сбалансированной схеме сверху-вниз проявляется как особый случай, когда схема проектирования оказалась вполне подходящей или задача мала по размеру... Хорошие проектировщики работают одновременно на нескольких уровнях абстракции и детализации" [8].

Текущий процесс продолжает выполняться на процессоре до тех пор, пока сам не уступит контроль над ним.

• Вытесняющее

Процесс с более высоким приоритетом, может отнимать процессор у исполняемого процесса с более низким приоритетом; обычно процессы с одинаковым приоритетом получают равные промежутки времени для выполнения, так что вычислительные ресурсы распределены справедливо.

• Невытесняющее

• Воспроизводимый	Организация в разумной степени управляет своими планами и обязательствами.
• Определенный	Процесс разработки в разумной степени определен, понятен и применяется на практике; он позволяет выбирать команду и предсказывать ход разработки. Следующая цель - оформить выработанную практику разработки как инструментальную среду.
• Управляемый	Организация выработала количественные показатели процесса. Цель состоит в снижении затрат на сбор данных и налаживание механизмов обратной связи, позволяющих данным влиять на процесс.
• Оптимальный	Организация имеет отлаженный процесс, устойчиво выдающий результаты высокого качества, своевременно, предсказуемо и эффективно.

К сожалению, как отмечают Парнас и Клеменс: "Мы никогда не отыщем процесс, который дал бы нам возможность проектировать программы строго рациональным образом", поскольку дело это творческое и новаторское по определению. Однако, продолжают они, "хорошей новостью является, то, что мы можем его имитировать... (Поскольку) разработчики нуждаются в руководстве, мы приблизимся к рациональной разработке, если будем следовать процессу, а не действовать, как попало. Когда организация занята многими программными продуктами, есть смысл иметь стандартную процедуру... Если мы держим в голове идеальный процесс, становится легче измерять успехи проекта" [9].

С приобретением опыта у организации встает вопрос: "Как примирить творчество и новации с возрастающей управляемостью?". Ответ состоит в разграничении макро- и микроэлементов процесса проектирования. Микропроцесс родственен спиральной модели развития, предложенной Боемом, и служит каркасом для итеративного подхода к развитию [10]. Макропроцесс близок к традиционному "водопаду" и задает направляющие рамки для микропроцесса. Примиряя эти два в корне различных процесса, мы имитируем полностью рациональный процесс разработки и обретаем основу для определенного уровня зрелости в деле создания программного обеспечения.

Мы должны подчеркнуть, что каждый проект уникален, и, следовательно, разработчик сам должен поддерживать баланс между неформальностью микропроцесса и формальностью макропроцесса. Для исследовательских приложений, разрабатываемых тесно сплоченной командой высококвалифицированных разработчиков, чрезмерная формальность негативно отразится на новациях; для очень сложных проектов, разрабатываемых большим коллективом разработчиков, отделенных друг от друга пространством и временем, недостаток формальности приводит к хаосу.

Оставшаяся часть этой главы дает обзор и детальное описание целей, результатов, видов деятельности и измеримых характеристик, составляющих микро- и макропроцессы разработки. В следующей главе мы рассмотрим практические проявления этих процессов, в первую очередь с точки зрения менеджеров, которые должны надзирать за ходом объектно-ориентированного проекта.

6.2. Микропроцесс проектирования

Обзор

Микропроцесс объектно-ориентированной разработки приводится в движение потоком сценариев и архитектурных продуктов, которые порождаются и последовательно уточняются в макропроцессе. Микропроцесс, по большей части, - повседневный труд отдельного разработчика или небольшого коллектива разработчиков.

Микропроцесс относится в равной степени к программисту и архитектору программной системы. С точки зрения программиста, микропроцесс предлагает руководство в принятии бесчисленного числа ежедневных тактических решений, которые являются частью процесса создания и подгонки архитектуры системы. С точки зрения архитектора, микропроцесс является основой для развития архитектуры и опробования альтернатив.

В микропроцессе традиционные фазы анализа и проектирования умышленно перемешаны, а управление осуществляется "по возможности". Как отмечает Стра-уструп, "не существует рецептов, которые могли бы заменить ум, опыт и хороший вкус в проектировании и программировании... Различные фазы программного проекта, такие, как проектирование, программирование и тестирование, неотделимы друг от друга" [II].

Как показано на рис. 6-1, микропроцесс обычно состоит из следующих видов деятельности:

- выявление классов и объектов на данном уровне абстракции
- выяснение семантики этих классов и объектов
- выявление связей между этими классами и объектами
- спецификация интерфейса и реализация этих классов и объектов.

Теперь рассмотрим каждый из этих видов деятельности подробно.

Выявление классов и объектов

Цель. Цель выявления классов и объектов состоит в том, чтобы найти границы предметной области. Кроме того, эта деятельность является первым шагом в продумывании объектно-ориентированной декомпозиции разрабатываемой системы.

Мы применяем этот шаг в анализе, когда обнаруживаем абстракции, составляющие словарь предметной области и ограничиваем нашу задачу, решая, что важно, а что - нет. Такие действия необходимы при проектировании, когда мы изобретаем новые абстракции, которые являются

составными частями решения. Переходя к программной реализации, мы применяем процедуру выявления, чтобы изобрести простые абстракции, из которых строятся более сложные, и обнаружить общие черты существующих абстракций, дабы упростить архитектуру системы.

Результаты. Главным результатом этого шага является обновляющийся по мере развития проекта словарь данных. Вначале достаточно составить список действующих лиц, состоящий из всех заметных классов и объектов, названых именами, отражающими их смысл [12]. Когда словарь разрастется, можно сделать простейшую базу данных, или более специальный инструмент проектирования, непосред-

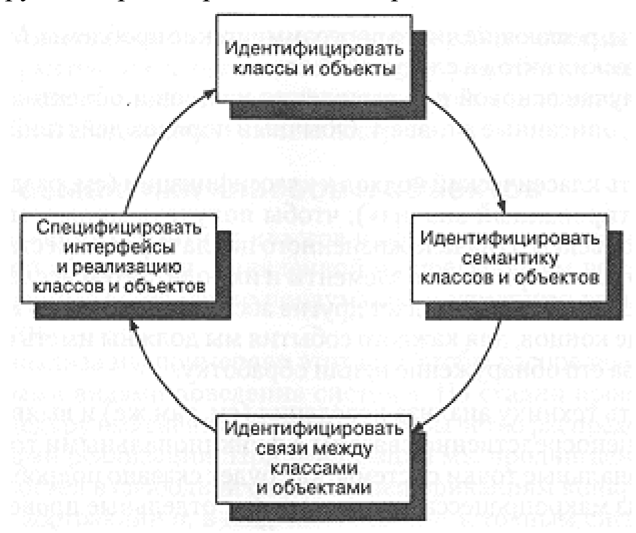


Рис. 6-1. Микропроцесс

ственно поддерживающий выбранный метод разработки.[19] В своих более формальных разновидностях словарь данных служит предметным указателем для всех остальных компонентов проекта, включая диаграммы и спецификации обозначений объектно-ориентированного проектирования.

Таким образом, словарь данных - центральное хранилище относящихся к системе абстракций. Вначале допустимо держать словарь данных открытым для изменений: некоторые персонажи могут оказаться классами, некоторые - объектами, другие - атрибутами, а иные - просто синонимами других абстракций. Постепенно содержимое словаря уточняется путем введения новых, исключения лишних и объединения схожих абстракций.

Создание словаря данных на этом шаге дает три существенных выигрыша. Во-первых, сама работа с ним помогает выработать общепринятую и исчерпывающую терминологию, которой можно пользоваться на протяжении всего проекта. Во-вторых, словарь - естественное оглавление ко всем материалам проекта и система точек входа для доступа к проекту в произвольном порядке. Это особенно полезно, когда в команду принимается новый разработчик, который должен быстро войти в курс дел. В-третьих, словарь данных позволяет архитектору окинуть весь проект единым взглядом, что может привести к открытию новых общностей, которые иначе могли бы быть упущены.

Виды деятельности. Как мы описывали в главе 4, выявление классов и объектов связано с двумя видами творческих актов: открытием и изобретением.

[19] Формально, словарь данных объектно-ориентированной разработки должен содержать спецификации каждого элемента архитектуры.

Не каждый член команды должен быть равно искусен во всем. Аналитики, особенно работающие с экспертами в предметной области, должны уметь хорошо обнаруживать абстракции, то есть находить осмысленные классы и объекты в предметной области. Тем временем архитекторы и старшие разработчики придумывают классы и объекты, решающие чисто программистские проблемы. Мы обсудим природу этих творческих актов в следующей главе.

В любом случае основой для выявления классов и объектов служат методы классификации, описанные в главе 4. Обычный порядок действий таков:

- Применить классический подход к классификации (см. раздел 4.2, "Объектно-ориентированный анализ"), чтобы получить множество кандидатов в классы и объекты. В начале жизненного цикла хорошими стартовыми точками являются материальные элементы и их роли. Затем исследовать последовательности событий, что даст другие абстракции первого и второго порядка: в конце концов, для каждого события мы должны иметь объект, который отвечает за его обнаружение и/или обработку.
- Применить технику анализа поведения (см. там же) и выявить абстракции, которые непосредственно связаны с функциональными точками системы. Функциональные точки системы, как будет сказано подробнее в этой главе, берутся из макропроцесса и представляют отдельные проверяемые и внешне наблюдаемые поведения системы. Как и в случае событий, для каждого поведения можно найти классы и объекты, которые инициируют его и участвуют в нем.
- Для соответствующих сценариев, созданных в макропроцессе, применить технику анализа вариантов (см. там же). В начале жизненного цикла мы исследуем самые общие сценарии поведения системы. В процессе разработки мы постепенно переходим ко все более детализированным сценариям, добираясь до самых темных уголков поведения системы.

В каждом из этих подходов CRC-карточки являются эффективным катализатором "мозгового штурма" и помогают теснее сплотить коллектив, подталкивая его членов к общению.[20]

Некоторые классы и объекты будут определены в начале жизненного цикла проекта неправильно, но это не всегда плохо. Многие осязаемые вещи и роли, которые мы перечислим в жизненном цикле, пройдут через весь путь вплоть до реализации - настолько они фундаментальны для нашей концептуальной модели. Разбираясь в задаче, мы, вероятно, будем изменять границы некоторых абстракций, перераспределяя ответственности, объединяя подобные или (чаще всего), разбивая большие абстракции на группы взаимодействующих, формируя таким образом некоторые механизмы решения.

Путевые вехи и характеристики. Мы благополучно завершим эту фазу, когда будем иметь достаточно стабильный словарь данных. Поскольку микропроцесс развивается итеративно, следует ожидать, что словарь будет закончен и закрыт лишь на очень поздней стадии проекта. Пока нас удовлетворяет обильный, даже избыточный набор абстракций с содержательными именами и разумным распределением обязанностей.

Признаком качества, следовательно, будет то, что словарь не подвергается серьезным изменениям каждый раз, когда мы проходим новую итерацию микропроцесса. Неустойчивость словаря показывает, что

[20] Это ужасно банально, но некоторые проектировщики программ и в самом деле не очень общительны.

разработчики еще не достигли желаемого, или в архитектуре что-то не так. По ходу разработки мы можем контролировать устойчивость нижних уровней архитектуры, отслеживая результаты локальных изменений взаимодействующих абстракций.

Выяснение семантики классов и объектов

Цель. Цель выяснения семантики классов и объектов - определить поведение и атрибуты каждой абстракции, выявленной на предыдущем шаге. При этом мы уточняем намеченные абстракции, продуманно и измеримо распределяя между ними обязанности.

На стадии анализа мы применяем этот шаг, чтобы распределить обязанности между различными видами поведения системы. На стадии проектирования мы применяем процедуру выяснения семантики, чтобы четко распределить обязанности между частями реализации. При реализации мы продвигаемся от описаний ролей и обязанностей в свободной форме к спецификациям конкретных протоколов для каждой абстракции и, в конечном счете, - к точным сигнатурам каждой операции.

Результаты. На этом шаге получаются несколько результатов. Первым является уточнение словаря данных, с помощью которого мы изначально присвоили обязанности абстракциям. В ходе проектирования мы можем выработать спецификации к каждой абстракции (как описано в главе 5), перечисляя имена операций в протоколе каждого класса. Затем, как можно скорее, мы выразим интерфейсы этих классов на языке реализации. Для C++ это означает создание .h-файлов, в Ada - спецификаций пакетов, в CLOS - обобщенных функций для каждого класса, в Smalltalk - это объявление, но не реализация методов каждого класса. Если проект связан с базой данных, особенно с объектно-ориентированной, на этом шаге мы получаем общий каркас нашей схемы данных.

В добавление к этим, по сути тактическим решениям, мы составляем диаграммы объектов и диаграммы взаимодействий, передающие семантику сценариев, создаваемых в ходе макропроцесса. Эти диаграммы формально отражают рас-кадровку каждого сценария и, таким образом, описывают явное распределение обязанностей среди взаимодействующих объектов. На этом шаге впервые появляются конечные автоматы для представления некоторых абстракций.

Чтобы команда разработчиков могла развивать согласованный язык обозначений и для учета обязанностей каждой абстракции, мы можем, как и на предыдущем шаге, использовать специализированную базу данных или другие, более специфические инструменты проектирования. Когда мы напишем на выбранном языке формальные интерфейсы классов, мы можем использовать наши инструменты проектирования для проверки и гарантии выполнения принятых решений.

Главная выгода большей формальности результатов на этом шаге состоит в том, что она помогает разработчику увидеть назначение всех протоколов абстракции. Невозможность четко определить смысл - признак зыбкости самих абстракций.

Виды деятельности. С этим шагом связано три вида деятельности: раскадровка, проектирование изолированных классов и поиск шаблонов.

Главными объектами раскадровки являются основные и второстепенные сценарии, полученные в макропроцессе. В ходе этой деятельности происходит нисходящее выяснение семантики. Там, где это касается функциональных точек системы, принимаются стратегические решения. Типичный ход выполнения действий может быть таким:

- Выбрать сценарий (или группу сценариев), связанный с отдельной функциональной точкой; на основании результатов предыдущего шага определить относящиеся к этому сценарию абстракции.
- Проследить действия в этом сценарии, наделяя каждую абстракцию обязанностями, достаточными, чтобы получить требуемое общее поведение. Если необходимо - выбрать атрибуты, которые будут представлять структурные элементы, требуемые для выполнения отдельных обязанностей.
- По ходу раскадровки перераспределить обязанности так, чтобы сбалансировать поведение. Где возможно, использовать или адаптировать уже существующие обязанности. Очень распространенным приемом является деление больших обязанностей на малые; иногда тривиальные обязанности объединяются в более сложные.

Неформально мы можем использовать для раскадровки CRC-карточки. Для большей формальности команде разработчиков следует составить диаграммы объектов и взаимодействий. На стадии анализа раскадровка обычно выполняется командой, включающей, как минимум, аналитика, эксперта в предметной области, архитектора и контролера качества. На стадии проектирования и позже, при реализации, раскадровка выполняется архитектором и старшими разработчиками для доводки стратегических решений, и отдельными разработчиками - для доводки тактических решений. Привлечение дополнительных членов команды к участию в раскадровке - в высшей степени эффективный путь обучения начинающих разработчиков и передачи им сложившегося видения архитектуры.

В начале разработки проекта мы можем задавать семантику классов и объектов в свободной форме, просто описывая обязанности каждой абстракции. Обычно достаточно фразы или предложения; если этого мало - мы встречаем верный признак того, что данная обязанность является чрезмерно сложной и должна разделиться на меньшие. На более поздних стадиях разработки, когда мы будем заниматься доводкой протоколов отдельных абстракций, можно указать имена специфических операций, не определяя их полные сигнатуры, которые мы выясним потом. Таким образом, мы получим соответствие: каждая обязанность выполняется набором операций, а каждая операция как-либо участвует в выполнении обязанностей соответствующей абстракции. После этого, чтобы отразить динамическую семантику протоколов классов,[21] имеющих управляемое событиями или зависящее от состояния поведение, мы можем построить конечные автоматы для них.

На этом шаге важно сосредоточить внимание больше на поведении, чем на структуре. Атрибуты представляют структурные элементы, а, значит, есть опасность, особенно на ранних стадиях анализа, преждевременным указанием некоторых атрибутов стеснить реализационные решения. Атрибуты должны идентифицироваться на этом этапе лишь настолько, насколько они необходимы в построении концептуальной модели сценария.

Проектирование изолированных классов - это восходящее выяснение семантики. Здесь мы концентрируем наше внимание на отдельных абстракциях и, применяя описанные в главе 3 эвристики для проектирования классов, рассматриваем их операции. Это действие по своей природе более тактическое, потому что здесь мы затрагиваем проектирование классов, а не архитектуры. Порядок его выполнения может быть следующим:

---

[21] Как мы описывали в главе 3, протокол определяет, что некоторые операции должны вызываться в определенном порядке. Для всех случаев кроме самых тривиальных операции редко встречаются в одиночестве; выполнение каждой из них имеет свои предусловия, проверка которых часто требует вызова других операции.

- Выбрать одну абстракцию и перечислить ее роли и обязанности.
- Определить необходимое множество операций, удовлетворяющих этим обязанностям. Попытаться, где возможно, использовать операции для концептуально схожих ролей и обязанностей повторно.
- Рассмотреть каждую операцию абстракции: если она не примитивна - выделить и определить примитивы. Составные операции могут быть оставлены в самом классе (либо из-за их общности, либо по соображениям эффективности) или могут быть отправлены в утилиту классов (если они будут часто изменяться). Где это возможно следует рассмотреть минимальный набор примитивных операций.
- Учесть конструирование, копирование и уничтожение объектов [13]. Если не имеется причин поступить иначе, лучше иметь общие стратегические принципы для таких операций, чем позволить отдельным классам вводить свои собственные решения.
- Придать завершенность: добавить другие примитивные действия, которые не нужны существующим клиентам, но "округляют" абстракцию, что повышает вероятность использования ее новыми клиентами. Помня, что невозможно иметь полную завершенность, стремиться к простоте.

Важно избегать преждевременного определения отношения наследования - это часто ведет к потере целостности типа.

На ранних этапах разработки проектировать отдельные классы можно изолировано. Однако, как только мы определим структуры наследования, этот шаг будет включать в себя размещение операций в иерархии классов. Рассматривая операции, связанные с некоторым уровнем абстракции, мы должны решить, на каком уровне абстракции их разместить. Операции, которые могут быть использованы несколькими классами одного уровня, должны быть помещены в их общий суперкласс, который, возможно, придется создать. Действия, которые совместно используются никак не связанными классами, должны быть инкапсулированы в класс-примесь.

Третий вид деятельности - поиск шаблонов - связан с обобществлением абстракций. Выявляя семантику классов и объектов, мы должны отмечать шаблоны поведения, которые могут пригодиться где-нибудь еще. Этот процесс может проистекать в следующем порядке:

- Имея полный набор сценариев на этом уровне абстракции, найти шаблоны взаимодействия абстракций. Такие взаимодействия могут представлять неявные идиомы или механизмы. Они должны быть исследованы, чтобы гарантировать, что не имеется никаких необоснованных различий в вызовах операций. Нетривиальные шаблоны взаимодействия нужно явно документировать как стратегические решения, чтобы они по возможности могли быть повторно использованы, а не изобретались заново. Это повышает архитектурную целостность.
- Имея набор обязанностей для данного уровня абстракции, отыскать шаблоны поведения. Общие роли и обязанности должны быть унифицированы в форме общих классов - базовых, абстрактных или примесей.
- Если уже специфицированы конкретные операции, найти шаблоны среди сигнатур операций. Если среди них встречаются часто повторяющиеся, устранить все непринципиальные различия и ввести классы-примеси или утилиты классов.

Выяснение и описание семантики применяется к категориям классов так же, как к отдельным классам. Семантика классов и их категорий определяет роли, обязанности и операции. Для отдельного класса операции могут быть со временем выражены как его функции-члены; в случае категории классов эти операции представляют экспортируемые из категории услуги, и в конечном счете реализуются набором сотрудничающих классов или отдельным классом. Таким образом, действия, описанные выше, применимы и к проектированию классов, и к проектированию архитектуры.

Путевые вехи и характеристики. Мы благополучно завершим этот шаг, когда будем иметь более или менее достаточный, примитивный и полный набор обязанностей и/или операций для каждой абстракции. В начале разработки достаточно иметь неформальный список обязанностей, а в дальнейшем мы постепенно уточняем семантику.

Качественные показатели включают все эвристики классов, описанные в главе 3. Сложные и туманные обязанности и операции говорят о том, что абстракции еще недостаточно определены. Невозможность написать конкретный файл заголовков или как-либо по другому формализовать интерфейс классов также говорит о том, что абстракции плохо сформулированы, или что основные понятия определяли не те люди.[22]

При просмотре сценариев ожидайте бурных дебатов. Это помогает разработчикам делиться архитектурными представлениями и развивать искусство определения абстракций. Не проверенные абстракции не стоит пытаться кодировать.

Выявление связей между классами и объектами

Цель. Цель выявления связей между классами и объектами - уточнить границы каждой обнаруженной ранее в микропроцессе абстракции и опознать все сущности, с которыми она взаимодействует. Это действие формализует концептуальное и физическое размежевание между абстракциями, начатое на предыдущем шаге.

Мы применяем этот шаг в анализе для спецификации связей между классами и объектами (включая некоторые важные отношения наследования и агрегации).

Существование ассоциации подразумевает некоторую семантическую зависимость между двумя абстракциями и возможность перехода от одной сущности к другой. Этот этап проектирования нужен, чтобы специфицировать взаимодействия, которые формируют механизмы нашей архитектуры и группирование классов в категории и модулей в подсистемы. В ходе реализации мы приводим ассоциации к более конкретному виду: инстанцирование, использование и т. д.

Результаты. Основными результатами этого шага являются диаграммы классов, объектов и модулей. Хотя в конце концов мы должны выразить наши решения, принятые при анализе и проектировании, на языке программирования, диаграммы дают более широкий обзор архитектуры и, кроме того, позволяют раскрыть отношения, которые с трудом формулируются на используемом языке реализации.

При анализе мы составляем диаграммы классов, на которых указываются ассоциации между абстракциями, и добавляем к ним детали, полученные на предыдущем шаге (операции и атрибуты некоторых абстракций), необходимые, чтобы передать суть наших решений. При проектировании мы уточняем эти диаграммы, чтобы отразить принятые

---

[22] Остерегайтесь аналитиков и архитекторов, если они не хотят или не могут выразить конкретно семантику своих абстракции; это признак надменности или беспомощности.

тактические решения о наследовании, агрегации, ин-станцировании и использовании.

Нет ни возможности, ни необходимости создавать исчерпывающий набор диаграмм, которые определили бы все возможные виды связей между нашими абстракциями. Нужно сосредоточиться на "интересных отношениях", причем подразумевается, что в число "интересных" входят те связи между абстракциями, которые отражают фундаментальные архитектурные решения или выражают детали, необходимые для реализации.

Результатом анализа на данном этапе являются диаграммы классов, которые содержат категории классов, идентифицирующие кластеры абстракций, сгруппированные по слоям и разделам. Эти результаты пригодятся и для документирования.

При анализе мы также строим диаграммы объектов, завершая тем самым просмотр сценариев, начатый на предыдущем шаге. Отличие в том, что мы можем теперь рассмотреть взаимодействия между классами и объектами и обнаружить скрытые ранее общие механизмы взаимодействия, которыми следует воспользоваться. Обычно это приводит к локальным перестройкам структуры наследования. При проектировании мы пользуемся диаграммами объектов вместе с более детализированным описанием состояний, чтобы показать действие наших механизмов в динамике. Явный результат этого шага - набор диаграмм, которые идентифицируют механизмы взаимодействия.

При реализации мы должны принять решения о физическом разбиении нашей системы на модули и о распределении процессов по процессорам. Эти решения мы можем выразить на диаграммах модулей и процессов.

На этом же шаге также обновляется словарь данных. В нем отражаются распределения классов и объектов по категориям и модулей по подсистемам.

Основная польза полученных результатов в том, что они помогают наглядно показать и понять отношения, которыми связаны концептуально и физически далекие сущности.

Виды деятельности. С этим шагом связано три вида деятельности: спецификация ассоциаций, идентификация различных взаимодействий и уточнение ассоциаций. Спецификация ассоциаций является одним из основных действий в анализе и на ранней стадии проектирования. Как объяснялось в главе 3, ассоциации семантически слабы: они обозначают только некоторую семантическую зависимость, роль каждого участника связи и кардинальность связи и, возможно, направление допустимого перехода. Однако для анализа и ранней стадии проектирования этого часто достаточно, ибо передаются все важные детали связей между двумя абстракциями, при этом предохраняя нас от поспешных решений о реализации. Типичный порядок выполнения данного этапа таков:

- Выбрать множество классов данного уровня абстракции или ассоциированных с некоторым набором сценариев; нанести на диаграммы все важнейшие операции и атрибуты, необходимые для иллюстрации существенных свойств моделируемой задачи.
- Выяснить наличие зависимости между каждыми двумя классами и установить ассоциацию, если она присутствует. Необходимость перехода от одного объекта к другому и неизбежность использования некоторого поведения другого объекта являются причиной введения ассоциации. Чтобы устранить косвенные зависимости, следует ввести новые абстракции, которые служили бы агентами или посредниками. Некоторые ассоциации могут быть сразу идентифицированы как отношение "частное/общее" или агрегации.

- Для каждой ассоциации определить роль каждого участника, если необходимо уточнить кардинальность и выявить другие ограничения.
- Проверить годность этих решений, для чего просмотреть сценарий и убедиться, что имеющиеся ассоциации необходимы и достаточны для получения требуемых переходов и поведения абстракций этого сценария.

Диаграммы классов - основные модели, получаемые на данном этапе. Идентификация взаимодействий происходит главным образом при проектировании и, как описано в главе 4, является задачей классификации. А, значит, она также требует творчества и интуиции. В зависимости от текущего состояния макропроцесса, мы должны рассмотреть несколько различных типов взаимодействия:

- Как часть формулировки наших стратегических решений, мы должны составить для каждого определенного на предыдущем шаге механизма диаграмму объектов, иллюстрирующую его динамическую семантику. Проверить каждый механизм в центральных и периферийных сценариях. Где возможен параллелизм, назначить объекты - актеры, агенты и серверы и способы синхронизации между ними. При этом может понадобиться ввести новые связи между объектами и устранить неиспользованные или избыточные.
- Если между классами наблюдается общность, необходимо поместить эти классы в иерархию "общее/частное". Как говорилось в главе 3, обычно лучше создать "лес" классов, чем единое дерево. На предыдущем шаге мы уже определили кандидатов на базовые, абстрактные классы и классы-примеси; теперь нужно разместить их в структуре наследования. Для существенных классов следует рассмотреть диаграммы классов и оценить их качество, согласно эвристикам главы 3. В частности, требует особого внимания иерархическая структура: она не должна быть слишком высокой или слишком короткой, чересчур широкой или узкой. Там, где встречаются шаблоны в структуре или поведении, нужно реорганизовать иерархию так, чтобы максимизировать общность (но не в ущерб простоте).
- Как часть архитектурного проектирования, мы должны рассмотреть группирование классов в категории и организацию модулей в подсистемы. Это - стратегические решения. Архитекторы могут использовать диаграммы классов, чтобы определить иерархию категорий классов, которая формирует слои и разделы разрабатываемой системы. Обычно это делается сверху вниз. Имея глобальное представление о системе, выделяют основные абстракции, выполняющие главные обязанности системы, которые являются логически связными и могут изменяться независимо. Архитектуру также можно модернизировать снизу вверх, когда при каждом прохождении через микропроцесс идентифицируются семантически замкнутые группы классов. Нужно также принять решения о распределении классов по категориям. Если существующие категории слишком раздуваются или обнаруживаются новые группы классов, можно ввести новые категории или реорганизовать старые. Выявление модулей (для физической модели системы) выполняется аналогично и принятые решения отражаются на диаграммах модулей.
- Распределение классов и объектов по модулям является до некоторой степени локальным решением и чаще всего отражает отношения видимости абстракций. Как мы указывали в главе 5, отображение логической модели в физическую дает возможность разработчику

открыть или ограничить доступ к каждой абстракции или упаковать вместе логически связанные абстракции, которые предполагается изменять по отдельности. Как мы обсудим в следующей главе, на отображение логической модели в физическую влияет также распределение обязанностей в команде проектировщиков. В любом случае все принятые решения можно выразить в виде диаграммы модулей.

Третий вид деятельности в этой фазе микропроцесса - уточнение ассоциа-ций - относится и к анализу, и к проектированию. При анализе мы можем провести вместо некоторых ассоциаций другие, семантически более точные связи, что-бы отразить наши достижения в понимании прикладной области. Таким образом, преобразовывая ассоциации и добавляя новые конкретные связи, мы готовим на-бросок реализации.

Отношения наследования, агрегации, инстанцирования и использования - важнейшие типы ассоциаций, представляющие для нас интерес вместе с такими свойствами, как метки, роли, кардинальность и т. д. Типичный порядок уточнения ассоциаций таков:

- Имея набор классов, уже разбитый на группы, следует найти шаблоны поведения, указывающие на возможную связь "общее/частное". Далее необходимо разместить эти классы в существующей структуре наследования или построить новую подходящую структуру.
- Если имеются шаблоны структуры, то, используя наследование с классами-примесями или агрегацию, попробовать ввести новые классы, отражающие общность структуры.
- Найти классы с похожим поведением, которые либо находятся на одном уровне, либо еще не входят в структуру наследования и рассмотреть возможность введения общих параметризованных классов.
- Рассмотреть существующие ассоциации с точки зрения переходов между ними и ограничить их насколько возможно. Если не требуется двустороннего перехода, считать связь простым отношением использования.
- Определить тактические детали: указать роли, ключи, кардинальность, дружественность и т. д. Не требуется излишне детализировать: достаточно включить лишь важные результаты анализа и проектирования или то, что необходимо для реализации.

Путевые вехи и характеристики. Мы благополучно завершим эту фазу, когда достаточно полно определим семантику и связи интересующих абстракций, чтобы приступить к началу реализации.

Меры качества - связность, зацепление и полнота. Пересматривая связи, которые мы обнаружили или изобрели в течение этой фазы, мы хотим получить связные и слабо зацепленные между собой абстракции. При этом мы должны идентифицировать все важные связи на данном уровне абстракции, чтобы реализация не требовала введения новых существенных связей или неестественного использования тех, которые мы уже определили. Если на следующем шаге обнаружится, что наши абстракции неудобны для реализации, то это будет признаком того, что мы еще не определили подходящего набора связей между ними.

Реализация классов и объектов

Цель. На этапе анализа реализация классов и объектов нужна, чтобы довести существующие абстракции до уровня, достаточного для обнаружения новых классов и объектов на следующем уровне абстракции; они сами будут в

дальнейшем поданы на новую итерацию микропроцесса. При проектировании целью реализации становится создание осязаемого представления наших абстракций путем выпуска последовательных исполнимых версий системы (макропроцесс).

Этот шаг намеренно выполняется позже всех, так как микропроцесс концентрирует внимание на поведении и откладывает насколько возможно решения о представлении. Такая стратегия оберегает разработчика от недозрелых решений, которые могут не оставить шансов на облегчение и упрощение архитектуры, и оставляет свободу выбора реализации (например, из соображений эффективности), гарантируя сохранение существующей архитектуры.

Результаты. На этом шаге мы принимаем решения о представлении каждой абстракции и об отображении этих абстракций в физическую модель. В начале процесса разработки мы формулируем эти тактические решения о представлении в форме уточненных спецификаций классов. Решения, имеющие общий интерес, или подходящие для повторного использования, мы документируем также на диаграммах классов (показывающих их статическую семантику), состояний и взаимодействия (показывающих их динамическую семантику). Когда становится ясно, на каком языке реализовывать проект, можно начинать программировать в псевдокоде или в исполнимом коде.

Чтобы раскрыть связи между логическим и физическим в нашей реализации системы, мы вводим диаграммы модулей, которые можно затем использовать, чтобы наглядно показать отображение нашей архитектуры в ее программную реализацию. Далее можно применить специфические инструментальные средства, которые позволяют либо генерировать код из диаграмм, либо восстанавливать диаграммы по реализации.

В этот шаг входит и обновление словаря данных, включая новые классы и объекты, которые были выявлены или изобретены при реализации существующих абстракций. Эти новые абстракции являются частью исходной информации для следующего цикла микропроцесса.

Виды деятельности. С реализацией связано одно главное действие: выбор структур и алгоритмов, которые представляют семантику определенных ранее микропроцессом абстракций. В отличие от первых трех стадий микропроцесса, сосредоточенных на внешних представлениях абстракций, этот этап акцентирует внимание на их внутреннем представлении.

На стадии анализа результаты этого действия относительно абстрактны: мы не так обеспокоены собственно реализацией, как заинтересованы в отыскании новых абстракций, которым можно делегировать обязанности. На стадии проектирования, особенно на поздних стадиях проектирования классов, мы действительно переходим к практическим решениям.

Типичный порядок действий таков:

- Пересмотреть протокол каждого класса. Идентифицировать стереотипы его использования объектами-клиентами, чтобы определить, какие операции являются центральными и, следовательно, должны быть оптимизированы. Для облегчения реализации разработать точные сигнатуры всех важнейших операций.
- Рассмотреть возможность использования параметризованных классов, закрытого или защищенного наследования в реализации. Выбрать подходящие классы-примеси или параметризованные классы (или создать новые, если задача достаточно общая) и соответствующим образом изменить структуру наследования.

- Рассмотреть объекты, которым можно делегировать обязанности. Для достижения эффективности может потребоваться незначительная реорганизация обязанностей и/или протокола абстракции нижнего уровня.
- Если семантика абстракции не может быть выражена через наследование, инстанцирование или делегирование, рассмотреть подходящее представление из примитивов языка. Выбрать то представление, которое оптимизирует стереотипы использования, учитывая важность операций с точки зрения объектов-клиентов абстракции. Однако помните, что невозможно оптимизировать каждый случай использования. Получив эмпирическую информацию из последовательных версий-прототипов, мы можем выделить абстракции, которые неэффективно используют время или память и улучшить их реализацию, не опасаясь нарушить предположения клиентов относительно нашей абстракции.
- Выбрать подходящий алгоритм для каждой операции. Ввести вспомогательные операции для расчленения сложных алгоритмов на более простые или более пригодные для повторного использования части. Рассмотреть возможные компромиссы, в частности, сделать выбор между хранением и вычислением отдельных членов-данных.

Путевые вехи и характеристики. На стадии анализа мы считаем, что благополучно завершили фазу реализации, когда идентифицировали все важные абстракции из тех, что необходимы для выполнения обязанностей абстракций, выявленных на этом цикле микропроцесса. На стадии проектирования реализация считается благополучно завершенной, когда мы получили исполнимую или почти исполнимую программную модель наших абстракций.

Главным показателем благополучия на этой фазе является простота. Сложные, неуклюжие или неэффективные реализации свидетельствуют о недостатках самой абстракции или о плохом ее представлении.

6.3. Макропроцесс проектирования

Обзор

Макропроцесс является контролирующим по отношению к микропроцессу. Макропроцесс предписывает ряд измеримых результатов и действий, которые позволяют команде разработчиков оценить риск, внести заблаговременные изменения в микропроцесс и сосредоточиться на коллективном анализе и проектировании. Макропроцесс - это деятельность всего коллектива в масштабе от недель до месяцев.

Многие элементы макропроцесса относятся к самой практике менеджмента программных проектов и поэтому выполняются одинаково, как для объектно-ориентированных, так и для других систем. Среди них - управление конфигурацией, гарантии качества, разбор программы и составление документации. В следующей главе мы рассмотрим эти практические вопросы в контексте объектно-ориентированного проектирования. Данная глава сосредоточена на описании специфики объектно-ориентированного подхода или (по определению Парнаса) на том, как мы уродуем рациональный процесс проектирования чтобы получить объектно-ориентированную систему.

Макропроцесс заботит в первую очередь технического руководителя команды разработчиков, цели которого несколько отличаются от задач отдельного разработчика. Они оба заинтересованы в качестве конечного

программного продукта, удовлетворяющем требованиям заказчика.[23] Однако, конечного пользователя мало волнует, правильно ли использованы в проекте параметризованные классы или полиморфизм; заказчик гораздо более обеспокоен сроками, качеством, полнотой и правильностью работы программы. Поэтому макропроцесс сконцентрирован на управлении риском и выявлении общей архитектуры - двух управляемых компонентах, имеющих решающее значение для сроков, полноты и качества проекта.

В макропроцессе в большой степени сохранены традиционные фазы анализа и проектирования и процесс в меру упорядочен. Как показано на рис. 6-2, макропроцесс обычно включает следующие действия:

- Выявление сущности требований к программному продукту (концептуализация).
- Разработка модели требуемого поведения системы (анализ).
- Создание архитектуры для реализации (проектирование).
- Итеративное выполнение реализации (эволюция).
- Управление эволюцией продукта в ходе эксплуатации (сопровождение).
-

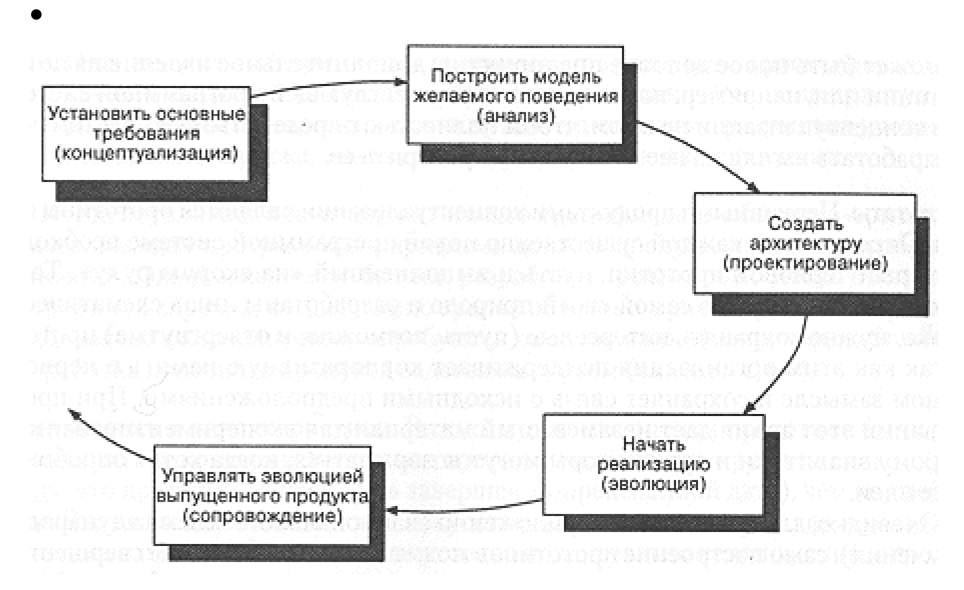


Рис. 6-2. Макропроцесс

У всех нетривиальных программных разработок макропроцесс продолжается и после создания и внедрения системы. Это особенно видно на примере организаций, специализирующихся на создании семейств программ, на которые часто выделяются значительные капиталовложения.

Основная философия макропроцесса состоит в постепенном развитии. Как его определяет Вонк, "при разработке методом последовательного развития, система выстраивается шаг за шагом, причем каждая новая версия содержит функциональность предыдущей, плюс новые функции" [14]. Этот подход чрезвычайно хорошо сочетается с объектно-ориентированной парадигмой и дает много возможностей для управления риском. Как утверждает Гилб: "Постепенная передача программ заказчику изобретена для того, чтобы заранее предупредить нас о надвигающихся неприятностях" [15].

Теперь детально рассмотрим каждое действие в макропроцессе. Естественно, одним из показателей зрелости организации, ведущей разработку, является знание случаев, когда надо обойти эти правила, что мы будем отдельно отмечать в нашем обзоре.

---

[23] Ну, конечно, не все, а большинство. К сожалению, некоторые менеджеры больше заинтересованы в развитии своей империи, чем в развитии программного продукта. Прибавьте к этому предыдущее примечание относительно аналитиков и проектировщиков. Я думаю, Данте мог бы найти для них подходящее место.

Концептуализация

Цель. Концептуализация должна установить основные требования к системе. Для каждой принципиально новой части программы или даже для нового применения существующей системы найдется такой момент, когда в голову разработчика, архитектора, аналитика или конечного пользователя западет идея о новом приложении.

Это может быть новое деловое предприятие, дополнительное изделие на поточной линии или, например, новая функция в существующей программной системе. Цель концептуализации не в том, чтобы полностью определить идею, а в том, чтобы выработать взгляд на нее и мысленно проверить ее.

Результаты. Первичными продуктами концептуализации являются прототипы системы. Определенно, каждой существенно новой программной системе необходим некоторый черновой прототип, пусть и выполненный "на скорую руку". Такие прототипы не полны по самой своей природе и разработаны лишь схематически. Однако, нужно сохранять интересные (пусть, возможно, и отвергнутые) прототипы, так как этим организация поддерживает корпоративную память о первоначальном замысле и сохраняет связь с исходными предположениями. При проектировании этот архив дает незаменимый материал для экспериментирования, к которому аналитики и архитекторы могут возвращаться, когда хотят опробовать новые идеи.

Очевидно, для грандиозных приложений (национального или международного значения), само построение прототипов может оказаться большим свершением. Ведь гораздо лучше столкнуться с трудностями при реализации, обнаружив, что неверны какие-то предположения о функциональности, эффективности, размере или сложности системы, чем пренебречь прогрессивным решением. Такое пренебрежение может грозить финансовой или социальной катастрофой.

Подчеркнем: прототипы хороши, но их следует выбросить. Нельзя позволять им непосредственно эволюционировать в готовую систему, если к этому не имеется достаточно серьезных оснований. Сжатые сроки не являются уважительной причиной: оптимизация краткосрочной разработки, игнорирующая последующие затраты владельца программного продукта, - типичный пример ложной экономии.

Виды деятельности. Концептуализация по самой своей природе - творческая деятельность, и, следовательно, она не должна быть скована жесткими правилами разработки. Возможно, самое важнее для организации - создать структуру, которая обеспечивала бы достаточные ресурсы для возникновения и исследования новых идей.[24] Новые идеи могут исходить из самых различных источников: конечных пользователей, групп пользователей, разработчиков, аналитиков, проектировщиков, распространителей и т. д. Для руководства важно вести регистрацию этих идей, располагая их по приоритетам и распределяя ограниченные ресурсы так, чтобы исследовать самые многообещающие из них. Когда для исследования выбрано конкретное направление, типичен следующий порядок дальнейших действий:

- Решить, какие цели преследуются при опробовании концепции и каковы критерии того, что считать благополучным исходом.
- Собрать подходящую команду для разработки прототипа. Часто она состоит из единственного члена (который и есть тот самый

[24] Если организация не сделает этого сама, то отдельные разработчики все равно сделают это, не спрашиваясь у компании, в которой они работают. Так и возникают новые программистские фирмы. Их появление хорошо для индустрии в целом, но не для самой осиротевшей компании.

мечтатель). Самое лучшее, что организатор может сделать, чтобы облегчить усилия команды - не стоять на ее пути.

- Оценить готовый прототип и принять ясное решение о проектировании конечного продукта или о дальнейшем исследовании. Решение приступить к разработке конечного продукта нужно принимать с разумным учетом потенциального риска, выявленного при опробовании концепции.

Концептуализация не содержит ничего специфически объектно-ориентированного. Каждая программная парадигма должна предусматривать опробование концепций. Однако, как часто бывает, разработка прототипов обычно происходит быстрее в тех случаях, когда на лицо зрелая объектно-ориентированная среда.

Довольно часто концепции опробуются на одном языке (например, на Smalltalk), а разработка конечного продукта ведется на другом (скажем, C++).

Путевые вехи и характеристики. Важно, чтобы для оценки прототипа были установлены четкие критерии. Работу над прототипом чаще планируют по срокам (имея в виду, что прототип должен быть завершен к определенной дате), чем по требованиям. Это не всегда плохо, так как искусственно ограничивает усилия по созданию прототипа и пресекает попытки выпустить концептуально недоношенный продукт.

Менеджеры верхнего звена могут оценить здоровье организации по ее отношению к новым идеям. Любая организация, которая сама не генерирует новые идеи, либо уже мертва, либо близка к этому. Наиболее благоразумное действие в такой ситуации - выделить независимые подразделения либо вообще уйти из бизнеса. С другой стороны, любая организация, заваленная новыми идеями, но неспособная определить их разумный приоритет, неуправляема. Такие компании часто тратят впустую существенные ресурсы, перескакивая к разработке изделия слишком рано, без исследования риска. Наиболее благоразумно здесь было бы формализовать процесс производства и наладить переход от концепции к продукту.

Анализ

Цель. Как утверждает Меллор, "цель анализа - дать описание задачи. Описание должно быть полным, непротиворечивым, пригодным для чтения и обозрения всеми заинтересованными сторонами, реально проверяемым" [16]. Говоря нашим языком, цель анализа - представить модель поведения системы.

Надо подчеркнуть, что анализ сосредоточен не на форме, а на поведении. На этой фазе неуместно заниматься проектированием классов, представлением или другими тактическими решениями. Анализ должен объяснить, что делает система, а не то, как она это делает. Любое, сделанное на стадии анализа (вопреки этому правилу) утверждение о том "как", может считаться полезным только для демонстрации поведения системы, а не как проверяемое требование к ее проектированию.

В этом отношении цели анализа и проектирования весьма различны. В анализе мы ищем модель мира, выявляя классы и объекты (их роли, обязанности и взаимодействия), которые формируют словарь предметной области. В проектировании мы изобретаем искусственные персонажи, которые реализуют поведение, требуемое анализом. В этом смысле, анализ - это деятельность, которая сводит вместе пользователей и разработчиков системы, объединяя их написанием общего словаря предметной области.

Сосредоточившись на поведении, мы приступаем к выяснению функциональных точек системы. Функциональные точки, впервые описанные Аланом Альбрехтом, обозначают видимые извне и поддающиеся проверке элементы поведения системы [17]. С точки зрения конечного пользователя,

функциональная точка представляет некоторое простейшее действие системы в ответ на некоторое событие.[25] Функциональные точки часто (но не всегда) обозначают отображение входов на выходы и таким образом представляют преобразования, совершаемые системой. С точки зрения аналитика, функциональные точки представляют кванты поведения. Действительно, функциональные точки - мера сложности системы: чем их больше, тем она сложнее. На стадии анализа мы передаем семантику функциональных точек сценариями.

Анализ никогда не происходит независимо. Мы не стремимся к исчерпывающему пониманию поведения системы и даже утверждаем, что сделать полный анализ до начала проектирования не только невозможно, но и нежелательно. Процесс построения системы поднимает вопросы о ее поведении, на которые реально нельзя дать гарантированный ответ, занимаясь только анализом. Достаточно выполнить анализ всех первичных элементов поведения системы и некоторого количества вторичных, добавляемых для гарантии того, что никакие существенные шаблоны поведения не пропущены.

Достаточно полный и формальный анализ необходим в первую очередь для того, чтобы ход проекта можно было проследить. Возможность проследить проект нужна для обеспечения возможности его просчитать, дабы гарантировать, что не пропущено ни одной функциональной точки. Возможность проследить проект является также основой управления риском. При разработке любой нетривиальной системы, менеджеры столкнутся с необходимостью сделать нелегкий выбор либо в распределении ресурсов, либо в решении некоторой тактической проблемы. Имея возможность проследить процесс от функциональных точек до реализации, гораздо легче оценить влияние подобных проблем на архитектуру.

Результаты. ДеШампо считает, что результатом анализа должно быть описание назначения системы, сопровождаемое характеристиками производительности и перечислением требуемых ресурсов [19]. В объектно-ориентированном проектировании мы получаем такие описания с помощью сценариев. Каждый сценарий представляет одну функциональную точку. Мы используем первичные сценарии для иллюстрации ключевого поведения и вторичные для описания поведения в исключительных ситуациях.

Как говорилось в предыдущих главах, мы используем технику CRC-карточек для раскадровки сценариев, а потом применяем диаграммы объектов для более точной иллюстрации семантики каждого сценария. Такие диаграммы должны демонстрировать взаимодействие объектов, обеспечивающее выполнение функций системы, и упорядоченный процесс этого взаимодействия, состоящий в посылке объектами сообщений друг другу. Кроме диаграмм объектов, в рассмотрение можно включить диаграммы классов (чтобы показать существующие ассоциации между классами объектов) и состояний (чтобы показать жизненный цикл важнейших объектов).

Часто эти результаты анализа объединяют в один формальный документ, который формулирует требования анализа к поведению системы, иллюстрируя их диаграммами, и показывает такие неповеденческие аспекты системы, как эффективность, надежность, защищенность и переносимость [20].

Побочным результатом анализа будет оценка риска: выявление опасных мест, которые могут повлиять на процесс проектирования. Обнаружение имеющегося риска в начале процесса проектирования облегчит возможные архитектурные компромиссы на поздних этапах разработки.

---

[25] Как отмечает Дрегер, в теории управления информационными системами функциональная точка представляет отдельную бизнес-функцию конечного пользователя [18].

Виды деятельности. С анализом связаны два основных вида деятельности: анализ предметной области и планирование сценариев.

Как мы описали в главе 4, анализ области должен идентифицировать обитающие в данной проблемной области классы и объекты. Прежде, чем взяться за разработку новой системы, обычно изучают уже существующие. В этом случае мы можем извлечь выгоду из опыта других проектов, в которых принимались сходные решения. Лучшим результатом анализа предметной области может явиться вывод, что нам не надо проектировать новый продукт, а следует повторно использовать или адаптировать существующую программу.

Планирование сценариев является центральным действием анализа. Интересно, что по этому вопросу, кажется, имеется совпадение мнений среди других методологов, особенно у Рубина и Голдберга (Rubin adn Goldberg), Адамса (Adams), Вирфс-Брока (Wirfs-Brock), Коада (Coad) и Джекобсона (Jacobson). Типичный порядок его выполнения следующий:

- Идентифицировать основные функциональные точки системы и, если возможно, сгруппировать функционально связанные виды поведения. Рассмотреть возможность создания иерархии функций, в которой высшие функции вытекают из низших.
- Для каждого представляющего интерес набора функциональных точек сделать раскадровку сценария, используя технику анализа поведения и примеров использования, описанную в главе 4.[26] В мозговом штурме каждого сценария эффективна техника CRC-карточек. Когда прояснится семантика сценариев, следует документировать их, используя диаграммы объектов, которые иллюстрируют объекты, инициирующие и обеспечивающие поведение, и их взаимодействие при выполнении действий сценария. Приложить описание событий, происходящих при выполнении сценария, и порядок выполняемых в результате действий. Кроме того, необходимо перечислить все предположения, ограничения и показатели эффективности для каждого сценария [21].
- Если необходимо, сделать вторичные сценарии, иллюстрирующие поведение системы в исключительных ситуациях.
- Для объектов с особо важным жизненным циклом описать диаграммы состояний (построить конечный автомат).
- Найти в сценариях повторяющиеся шаблоны и выразить их в терминах более абстрактных обобщенных сценариев или в терминах диаграмм классов, показывающих связи между ключевыми абстракциями.
- Внести изменения в словарь данных; включить в него новые классы и объекты, выявленные для каждого сценария, вместе с описанием их ролей и обязанностей.

Как описано в следующей главе, планирование сценариев выполняется аналитиками в сотрудничестве с экспертами в предметной области и архитекторами. В планировании сценария дополнительно должен участвовать контролер качества, так как сценарии представляют тестируемое поведение. Привлечение контролеров в самом начале процесса помогает сразу установить высокие стандарты качества. Эффективно также привлекать и других членов коллектива, чтобы дать им возможность включиться в процесс проектирования и ускорить понимание строения системы.

---

[26] Всесторонний анализ этого предмета можно найти в работах Джекобсона [22] и Рубина и Голдберга[23].

Путевые вехи и характеристики. Мы благополучно завершим эту фазу, когда мы будем иметь уточненные и подписанные сценарии для всех фундаментальных типов поведения системы. Говоря подписанные, мы предполагаем, что конечные результаты анализа проверялись экспертами, конечными пользователями, аналитиками и архитекторами; говоря фундаментальные, мы имеем в виду типы поведения, основные для данного приложения. Повторим, мы не ожидаем полного анализа,- достаточно рассмотреть только основные и несколько второстепенных видов поведения.

Степень совершенства анализа будет измеряться, в частности, его полнотой и простотой. Хороший анализ выявляет все первичные сценарии и, как правило, важнейшие вторичные. Разумный анализ включает также просмотр всех стратегически важных сценариев, так как это помогает привить единое видение системы всему коллективу разработчиков. Наконец, следует найти шаблоны поведения, которые давали бы возможно более простую структуру классов и учитывали бы все, что есть общего в различных сценариях.

Другой важной составной частью анализа является оценка риска, которая облегчит будущие стратегические и тактические компромиссы.

Проектирование

Цель. Цель проектирования - создать архитектуру развивающейся реализации и выработать единые тактические приемы, которыми должны пользоваться различные элементы системы. Мы начинаем процесс проектирования сразу после появления некоторой приемлемой модели поведения системы. Важно не начинать проектирование до завершения анализа. Равным образом важно избегать затягивания проектирования, пытаясь получить идеальную, а следовательно, недостижимую аналитическую модель.[27]

Результаты. Имеется два основных результата проектирования: описание архитектуры и выработка общих тактических приемов.

Мы можем описывать архитектуру путем построения диаграмм или создавая последовательные архитектурные релизы системы. Как описано в предыдущих главах, архитектура объектно-ориентированной системы выражает структуру классов и объектов в ней, поэтому можно использовать диаграммы классов и объектов, чтобы показать ее стратегическую организацию. Для описания архитектуры важно наглядно продемонстрировать группирование классов в категории классов (для логической архитектуры) и группирование модулей в подсистемы (для физической архитектуры). Можно распространять такие диаграммы, как часть формального документа, описывающего архитектуру, который должен быть доступен всем членам коллектива для ознакомления и внесения поправок при развитии архитектуры.

Мы используем архитектурные релизы системы как осязаемую демонстрацию строения архитектуры. Архитектурный релиз представляет собой как бы вертикальный разрез архитектуры, передающий важнейшую (но не полную) семантику существенных категорий и подсистем. Архитектурный релиз системы должен быть работающей программой, что позволяет измерять, изучать и оценивать архитектуру. Как мы увидим в следующем разделе, архитектурные релизы являются основой эволюции системы.

Общие тактические приемы - это локализованные механизмы, которые проявляются всюду в системе. К ним относятся такие аспекты проектирования, как принципы обнаружения и обработки ошибок, управление памятью, хранение и представление данных, подходы к управлению. Важно в явном виде описать эти приемы, чтобы не заставлять разработчиков

[27] Такая ситуация обычно классифицируется как паралич анализа.

отыскивать частные решения к общим задачам и не развалить нашу стратегическую архитектуру.

Мы описываем единые приемы в сценариях и действующих релизах каждого механизма.

Виды деятельности. С проектированием связано три действия: архитектурное планирование, тактическое проектирование и планирование релизов.

При архитектурном планировании мы занимаемся вертикальным и горизонтальным расчленением системы. Оно охватывает логическую декомпозицию, состоящую в группировании классов, и физическую декомпозицию, состоящую в разбиении на модули и назначении заданий процессорам. Типичный порядок действий таков:

- Рассмотреть группирование функциональных точек (найденных в анализе) и распределить их по слоям и разделам архитектуры. Функции базирующиеся одна на другой должны попасть в разные слои; функции, сотрудничающие между собой для обеспечения требуемого поведения системы на данном уровне абстракции должны попасть в разделы системы, представляющие услуги на этом уровне.
- Проверить архитектуру созданием действующих релизов, которые частично удовлетворяют семантике нескольких важнейших сценариев, предоставленных анализом.
- Оценить достоинства и недостатки архитектуры. Определить риск изменения каждого ключевого архитектурного интерфейса, чтобы можно было заранее распределить ресурсы при эволюции системы.

Архитектурное планирование сконцентрировано на том, чтобы создать в самом начале жизненного цикла каркас системы, а потом постепенно развивать его.

Тактическое проектирование состоит в принятии решений о множестве общих приемов. Как описано ранее в этой главе, плохое тактическое проектирование может разрушить даже очень продуманную архитектуру. Мы можем уменьшить этот риск, явно выделив тактические приемы и решив твердо их придерживаться. Типичный порядок действий таков:

- Перечислить все случаи, когда нужно следовать единым общим приемам. Некоторые из них окажутся фундаментальными, независимыми от предметной области, например, управление памятью, обработка ошибок и т. д. Другие будут специфичны для данной области и будут содержать свойственные этой области идиомы и механизмы, такие, как принципы управления системами реального времени или транзакциями и базами данных в информационных системах.
- Для каждого приема составить сценарий, описывающий его семантику. Затем выразить ее в виде исполнимого прототипа, который может быть уточнен и представлен инструментально.
- Документировать каждый принцип и распространить полученные документы, чтобы обеспечить единое архитектурное видение.

Программные релизы закладывают основы архитектурной эволюции системы. По полученным на стадии анализа функциональным точкам и оценкам риска, релизы выпускаются со все более широкими функциональными возможностями и, в конечном счете, достигают требований, предъявляемых к конечной системе. Типичный порядок действий таков:

- Полученные в результате анализа сценарии упорядочить от основных к второстепенным. Приоритетность сценариев лучше выяснить

вместе с экспертом в предметной области, аналитиком, архитектором и контролером качества.

- Распределить функциональные точки по релизам так, чтобы последний ре-лиз в серии представлял результирующую систему.
- Определить цели и расписание релизов так, чтобы дать время на разработку и синхронизировать релизы с другими действиями, например, с разработкой документации и полевыми испытаниями.
- Начать планирование задач, учитывая критические места проекта и ресурсы, отведенные на выпуск каждого релиза.

Естественным побочным результатом планирования релизов является план, в котором определены расписание работ, задачи коллектива и оценка риска.

Путевые вехи и характеристики. Мы благополучно закончим эту фазу, когда получим проверенную и утвержденную архитектуру, прошедшую прототипирова-ние и формализованные обзоры. Кроме этого, должны быть утверждены все важные тактические приемы и план последовательных релизов.

Основным признаком совершенства является простота. Хорошая архитектура имеет характеристики организованной сложной системы (см. главу 1).

Главные выгоды от этой деятельности - раннее выявление архитектурных просчетов и утверждение единых приемов, которые позволяют получить более простую архитектуру.

Эволюция

Цель. Цель эволюции - наращивать и изменять реализацию, последовательно совершенствуя ее, чтобы в конечном счете создать готовую систему.

Эволюция архитектуры в значительной степени состоит в попытке удовлетворить нескольким взаимоисключающим требованиям ко времени, памяти и т. д. - одно всегда ограничивает другое. Например, если критичен вес компьютера (как при проектировании космических систем), то должен быть учтен вес отдельного чипа памяти. В свою очередь количество памяти, допустимое по весу, ограничивает размер программы, которая может быть загружена. Ослабьте любое ограничение, и станет возможным альтернативное решение; усильте ограничение, и некоторые решения отпадут. Эволюция при реализации программного проекта лучше чем монолитный набор приемов помогает определить, какие ограничения существенны, а какими можно пренебречь. По этой причине эволюционная разработка сосредоточена прежде всего на функциональности и только затем - на локальной эффективности. Обычно в начале проектирования мы слишком мало знаем, чтобы предвидеть слабое место в эффективности системы. Анализируя поведение каждого нового релиза, используя гистограммы и тому подобную технику, команда разработчиков через какое-то время сможет лучше понять, как настроить систему.

Таким образом, эволюция - это и есть процесс разработки программы. Как пишет Андерт, проектирование "есть время новшеств, усовершенствований, и неограниченной свободы изменять программный код, чтобы достигнуть целей. Производство - управляемый методичный процесс подъема качества изделия к надлежащему уровню" [24].

Пейдж-Джонс называет ряд преимуществ такой поступательной разработки:

- "Обеспечивается обратная связь с пользователями, когда это больше всего необходимо, полезно и значимо.

- Пользователи получают несколько черновых версий системы для сглаживания перехода от старой системы к новой.
- Менее вероятно, что проект будет снят с финансирования, если он вдруг выбился из графика.
- Главные интерфейсы системы тестируются в первую очередь и наиболее часто.
- Более равномерно распределяются ресурсы на тестирование.
- Реализаторы могут быстрее увидеть первые результаты работы системы, что их морально поддерживает.
- Если сроки исполнения сжатые, то можно приступить к написанию и отладке программ до завершения проектирования".

Результаты. Основным результатом эволюции является серия исполнимых релизов, представляющих итеративные усовершенствования изначальной архитектурной модели. Вторичным продуктом следует признать выявление поведения, которое используется для исследования альтернативных подходов и дальнейшего анализа темных углов системы.

Действующие релизы выпускаются по графику, намеченному в начале планирования. Для скромного по размерам проекта, требующего 12-18 месяцев на разработку от начала до конца, это могло бы означать: по релизу каждые два или три месяца. Для более сложных проектов, требующих больше усилий разработчиков, можно выпускать релиз каждые шесть месяцев и реже. Более редкий график подозрителен, так как он не вынуждает разработчиков должным образом завершать микропроцессы и может скрыть опасные области.

Для кого делается действующий релиз программы? В начале процесса разработки основные действующие релизы передаются разработчиками контролерам качества, которые тестируют их по сценариям, составленным при анализе, и накапливают информацию о полноте, корректности и устойчивости работы релиза. Это раннее накопление данных помогает при выявлении проблем качества, которые будут учтены в следующих релизах. Позднее действующие релизы передаются конечным (альфа и бета) пользователям некоторым управляемым способом. "Управляемым" означает, что разработчики тщательно выверяют требования к каждому релизу и определяют аспекты, которые желательно проверить и оценить.

Специфика микропроцесса предполагает, что при многочисленных внутренних релизах разработчики выпускают наружу лишь некоторые исполнимые версии. Внутренние релизы представляют своего рода процесс непрерывной интеграции системы и завершают каждый цикл микропроцесса.

Косвенно подразумевается, что документация системы эволюционирует вместе с архитектурными релизами. Чтобы не относиться к ведению документации как к основному занятию, лучше всего получать ее, как естественный, полуавтоматически генерируемый побочный продукт эволюционного процесса.

Виды деятельности. Эволюция связана с двумя видами деятельности: микропроцесс и управление изменениями.

Работа, выполняемая между релизами, представляет процесс разработки в сжатом виде: это как раз и есть один цикл микропроцесса. Мы начинаем с анализа требований к следующему релизу, переходим к проектированию архитектуры и исследуем классы и объекты, необходимые для реализации этого проекта. Типичный порядок действий таков:

- Определить функциональные точки, которые попадут в новый релиз, и области наивысшего риска, особенно те, которые были выявлены еще при эволюции предыдущего релиза.

- Распределить задачи по релизам среди членов команды и начать новый микропроцесс. Контролировать микропроцесс, просматривая проект, и проверять состояние дел в важных промежуточных этапах с интервалами от нескольких дней до двух недель.
- Когда потребуется понять семантику требуемого поведения системы, поручить разработчикам сделать прототип поведения. Четко установить назначение каждого прототипа и определить критерии готовности. После завершения решить, как включить результаты прототипирования в этот или последующие релизы.
- Завершить микропроцесс интеграцией и очередным действующим релизом.

После каждого релиза следует перепроверить сроки и требования в основном плане релизов. Как правило, это незначительные корректировки дат или перенос функциональности из одного релиза в другой.

Управление изменениями необходимо именно в связи со стратегией итеративного развития. Всегда соблазнительно вносить неупорядоченные изменения в иерархию классов, их протоколы или механизмы, но это подтачивает стратегическую архитектуру и приводит к тому, что разработчики сами начинают путаться в собственном коде.

При эволюции системы на практике ожидаются следующие типы изменений:

- Добавление нового класса или нового взаимодействия между классами.
- Изменение реализации класса.
- Изменение представления класса.
- Реорганизация структуры классов.
- Изменение интерфейса класса.

Каждый тип изменений имеет свою причину и стоимость.

Проектировщик вводит новые классы, если обнаружились новые абстракции или понадобились новые механизмы. Цена выполнения таких изменений обычно несущественна для управления разработкой. Если добавляется новый класс, нужно рассмотреть, куда он попадет в существующей структуре классов. Когда вводится новое взаимодействие классов, должен быть произведен минимальный анализ предметной области, чтобы убедиться, что оно действительно удовлетворяет одному из шаблонов взаимодействия.

Изменение реализации также обходится недорого. Обычно при объектно-ориентированной разработке сначала создается интерфейс класса, а потом пишется его реализация (то есть код функций-членов). Если только интерфейс в приемлемой степени стабилен, можно выбрать любое внутреннее представление этого класса и выполнить реализацию его методов. Реализация отдельного метода может быть изменена (обычно для исправления ошибки или повышения эффективности) позже. Можно скорректировать реализацию метода, чтобы воспользоваться преимуществами новых методов, определенных в существующем или во вновь введенном суперклассе. В любом случае изменение реализации метода обходится сравнительно недорого, особенно, если она была своевременно инкапсулирована.

Подобным образом можно было бы изменить представление класса (в C++ - защищенные и закрытые члены класса). Обычно это делается, чтобы получить более эффективные (с точки зрения памяти или скорости) экземпляры класса. Если представление класса инкапсулировано, что возможно в Smalltalk, C++, CLOS и Ada, то изменение в представлении не будет разрушать логику взаимодействия объектов-пользователей с

экземплярами класса (если, конечно, новое представление обеспечивает ожидаемое поведение класса). С другой стороны, если представление класса не инкапсулировано, что также возможно в любом языке, то изменение в представлении класса чрезвычайно опасно, так как клиенты могут от него зависеть. Это особенно верно в случае подклассов: изменение представления суперкласса вызовет изменения представления всех его подклассов. Во всяком случае, изменение представления класса имеет цену: нужно произвести перекомпиляцию интерфейса и реализации класса, сделать то же для всех его клиентов, для клиентов тех клиентов и т. д.

Реорганизация структуры классов системы встречается довольно часто, хотя и реже, чем другие упомянутые виды изменений. Как отмечают Стефик и Бобров, "Программисты часто создают новые классы и реорганизуют имеющиеся, когда они видят удобную возможность разбить свои программы на части" [26]. Изменение структуры классов обычно происходит в форме изменения наследственных связей, добавления новых абстрактных классов и перемещения обязанностей и реализации общих методов в классы более верхнего уровня в иерархии классов. На практике структура классов системы особенно часто реорганизуется вначале, а потом, когда разработчики лучше поймут взаимодействие ключевых абстракций, стабилизируется. Реорганизация структуры классов поощряется на ранних стадиях проектирования, потому что в результате может получиться более лаконичная программа. Однако реорганизация структуры классов не обходится даром. Обычно изменение положения верхнего класса в иерархии делает устаревшими определения всех классов под ним и требует их перекомпиляции (а, значит, и перекомпиляции всех зависимых от них классов и т. д.).

Еще один важный вид изменений, к которому приходится прибегать при эволюции системы, - изменение интерфейса класса. Разработчик обычно изменяет интерфейс класса, чтобы добавить некоторый новый аспект, удовлетворить семантике некоторой новой роли объектов класса или добавить новую операцию, которая всегда была частью абстракции, но раньше не была экспортирована, а теперь понадобилась некоторому объекту-пользователю. На практике использование эвристик для построения классов, которые мы обсуждали в главе 3 (особенно требование примитивного, достаточного и полного интерфейса), сокращает вероятность таких изменений. Однако наш опыт никогда не бывает окончательным. Мы никогда не определим нетривиальный класс так, чтобы интерфейс его сразу оказался правильным.

Редко, но встречается удаление существующего метода; это обычно делается только для того, чтобы улучшить инкапсуляцию абстракции. Чаще мы добавляем новый метод или переопределяем метод, уже объявленный в некотором суперклассе. Во всех трех случаях это изменение дорого стоит, потому что оно логически затрагивает всех клиентов, требуя как минимум их перекомпиляции. К счастью, эти последние виды изменений, добавление и переопределение методов, совместимы снизу вверх. На самом деле вы обнаружите, что большинство изменений интерфейса, произведенного над определенными классами при эволюции системы, совместимы снизу вверх. Это позволяет для уменьшения воздействия этих изменений применить такие изощренные технологии, как инкрементная компиляция. Инкрементная компиляция позволяет нам вместо целых модулей перекомпилировать только отдельные описания и операторы, то есть перекомпиляции большинства клиентов можно избежать.

Почему перекомпиляция так неприятна? Для маленьких систем здесь нет проблем: перекомпиляция всей системы занимает несколько минут. Однако для больших систем это совсем другое дело. Перекомпиляция программы в сотни тысяч строк может занимать до половины суток машинного времени. Представьте себе, что вам понадобилось внести изменение в программное обеспечение компьютерной системы корабля. Как

вы сообщите капитану, что он не может выйти в море, потому что вы все еще компилируете? В некоторых случаях цена перекомпиляции бывает так высока, что разработчикам приходится отказаться от внесения некоторых, представляющих разумные усовершенствования, изменений. Перекомпиляция представляет особую проблему для объектно-ориентированных языков, так как наследование вводит дополнительные компиляционные зависимости [27]. Для строго типизированных объектно-ориентированных языков программирования цена перекомпиляции может быть даже выше; в этих языках время компиляции принесено в жертву безопасности.

Все изменения, обсуждавшиеся до настоящего времени, сравнительно легкие:

самый большой риск несут существенные изменения в архитектуре, которые могут погубить весь проект. Часто такие изменения производят чересчур блестящие инженеры, у которых слишком много хороших идей [28].

Путевые вехи и характеристики. Мы благополучно завершим фазу реализации, когда релизы перерастут в готовый продукт. Первой мерой качества, следовательно, будет то, в какой степени мы справились с реализацией функциональных точек, распределенных по промежуточным релизам, и насколько точно соблюдается график, составленный при их планировании.

Две других основных меры качества - скорость обнаружения ошибок и показатель изменчивости ключевых архитектурных интерфейсов и тактических принципов.

Грубо говоря, скорость обнаружения ошибок - это мера того, как быстро отыскиваются новые ошибки [29]. Вкладывая средства в контроль качества в начале разработки, мы можем получить количественные оценки качества для каждого ре-лиза, которые менеджеры команды смогут использовать для определения областей риска и обновления команды разработчиков. После каждого релиза должен наблюдаться всплеск обнаружения ошибок. Стабильность этого показателя обычно свидетельствует о том, что ошибки не обнаруживаются, а его чрезмерная величина говорит о том, что архитектура еще не стабилизировалась или что новые элементы неверно спроектированы или реализованы. Эти характеристики используются при уточнении цели очередного релиза.

Показатель изменчивости архитектурного интерфейса или тактических принципов является основной характеристикой стабильности архитектуры [30]. Локальные изменения вероятны в течение всего процесса эволюции, но если структуры наследования или границы между категориями классов или подсистем постоянно перестраиваются, то это признак нерешенных проблем в архитектуре, что должно быть учтено как область риска при планировании следующего релиза.

Сопровождение

Цель. Сопровождение - это деятельность по управлению эволюцией продукта в ходе его эксплуатации. Она в значительной степени продолжает предыдущие фазы, за исключением того, что вносит меньше архитектурных новшеств. Вместо этого делаются более локализованные изменения, возникающие по мере учета новых требований и исправления старых ошибок.

Леман и Белади сделали несколько неоспоримых наблюдений, рассматривая процесс "созревания" уже внедренной программной системы:

- "Эксплуатируемая программа должна непрерывно изменяться; в противном случае она будет становиться все менее и менее полезной (закон непрерывного изменения).

- Когда эволюционирующая программа изменяется, ее структура становится более сложной, если не прилагаются активные усилия, чтобы этого избежать (закон возрастающей сложности)" [3].

Мы отличаем понятие сохранения системы программного обеспечения от ее сопровождения. При сопровождении разработчики вносят непрерывные усовершенствования в существующую систему; сопровождением обычно занимается другая группа людей, отличная от группы разработчиков. Сохранение же основано на привлечении дополнительных ресурсов для поддержания устаревшей системы (которая часто имеет плохо разработанную архитектуру и, следовательно, трудна для понимания и модификации). Итак, нужно принять деловое решение: если цена владения программным продуктом выше, чем цена разработки новой системы, то наиболее гуманный образ действий - оставить старую систему в покое или покончить с ней.

Результаты. Поскольку сопровождение является в определенном смысле продолжением эволюции системы, ее результаты похожи на то, чего мы добивались на предыдущих этапах. В дополнение к ним, сопровождение связано также с управлением списком новых заданий. Кроме тех требований, которые по каким-либо причинам не были учтены, вероятно, уже вскоре после выпуска работающей системы, разработчики и конечные пользователи обменяются множеством пожеланий и предложений, которые они хотели бы увидеть воплощенными в следующих версиях системы. Заметим, что когда с системой поработает больше пользователей, выявятся новые ошибки и неожиданные методы использования, которых не смогли предвидеть контролеры качества.[28] В список заносятся обнаруженные дефекты и новые требования, которые будут учтены при планировании новых релизов в соответствии с их приоритетом.

Виды деятельности. Сопровождение несколько отличается от эволюции системы. Если первоначальная архитектура удалась, добавление новых функций и изменение существующего поведения происходят естественным образом.

Кроме обычных действий по эволюции, при сопровождении нужно определить приоритеты задач, собранных в список замечаний и предложений. Типичный порядок действий таков:

- Упорядочить по приоритетам предложения о крупных изменениях и сообщения об ошибках, связанных с системными проблемами, и оценить стоимость переработки.
- Составить список этих изменений и принять их за функциональные точки в дальнейшей эволюции.
- Если позволяют ресурсы, запланировать в следующем релизе менее интенсивные, более локализованные улучшения.
- Приступить к разработке следующего эволюционного релиза программы.

Путевые вехи и характеристики. Путевыми вехами сопровождения являются продолжающееся производство эволюционирующих релизов и устранение ошибок.

Мы считаем, что занимаемся именно сопровождением системы, если архитектура выдерживает изменения; мы определим, что вошли в стадию сохранения, когда количество ресурсов, требуемых для достижения нужного улучшения, начнет резко нарастать.

---

[28] Пользователи проявляют чудеса изобретательности в использовании системы самым необычным образом.

Выводы

- Удачные проекты обычно характеризуются ясным представлением об архитектуре и хорошо управляемым итеративным жизненным циклом.
- Идеально рациональный процесс проектирования невозможен, но его можно имитировать, сочетая микро- и макропроцесс разработки.
- Микропроцесс объектно-ориентированной разработки приводится в движение потоком сценариев и продуктов архитектурного анализа (макропроцесс); микропроцесс представляет ежедневную деятельность команды разработчиков.
- Первый шаг в микропроцессе связан с идентификацией классов и объектов на данном уровне абстракции; основными видами деятельности являются открытие и изобретение.
- Второй шаг микропроцесса состоит в выявлении семантики классов и объектов; основными видами деятельности здесь являются раскадровка сценариев, проектирование изолированных классов и поиск шаблонов.
- Третий шаг микропроцесса - выявление связей между классами и объектами; основными действиями являются спецификация ассоциаций, выявление взаимодействий и уточнение ассоциаций.
- Четвертый шаг микропроцесса связан с реализацией классов и объектов; основное действие - выбор структур данных и алгоритмов.
- Макропроцесс объектно-ориентированной разработки управляет микропроцессом, определяет измеримые характеристики проекта и помогает контролировать риск.
- Первый шаг макропроцесса - концептуализация, которая устанавливает основные требования к системе; она служит для опробования концепций и, по большей части, не должна контролироваться, чтобы предоставить неограниченную свободу фантазии.
- Второй шаг макропроцесса - анализ. Его цель - получить модель поведения системы. Основными действиями на этом этапе являются анализ предметной области и планирование сценариев.
- Третий шаг макропроцесса - проектирование. На этом шаге создается архитектура реализации и вырабатываются единые тактические приемы; основными действиями являются архитектурное планирование, тактическое проектирование и планирование релизов.
- Четвертый шаг макропроцесса - эволюция, последовательно приближающая систему к желаемому результату. Основные действия - применение микропроцесса и управление изменениями.
- Пятый шаг макропроцесса - сопровождение, то есть управление эволюцией системы в ходе ее эксплуатации; основные действия похожи на действия предыдущего шага, но к ним добавляется работа со списком улучшений и исправлений.

Дополнительная литература

Ранняя форма процесса, описанного в этой главе, была впервые опубликована Бучем (Booch) [F 1982]. Берард (Berard) позднее развил эту работу в статье [F 1986]. Среди родственных подходов можно назвать GOOD

(General Object-Oriented Design) Сейдвица и Старка (Seidewitz and Stark) [F 1985,1986,1987], SOOD (Structured Object-oriented Design) корпорация Локхид (Lockheed) [C 1988], MOOD (Multipleview Object-oriented Design) Керта (Kerth) [F 1988], и HOOD (Hierarchical Object-oriented Design), предложенный CISI Ingenierie и Matra для европейской космической станции [F 1987]. Более свежие ссылки: Страуструп (Stroustrup) [G 1991] и Microsoft [G 1992], где предложены сходные процессы.

В дополнение к работам, упомянутым в дополнительной литературе к главе 2, ряд других методологов предложил специфические процессы объектно-ориентированного развития. На эти работы есть много библиографических ссылок. Вот наиболее интересные из них: Алабио (Alabios) [F 1988], Бойд (Boyd) [F 1987], Бур (Buhr) [F 1984], Черри (Cherry) [F 1987,1990], деШампо (deChampeaux) [F 1992], Фелсингер (Felsinger) [F 1987], Файерсмит (Firesmith) [F 1986,1993], Хайнс и Юнгер (Hines and Unger) [G 1986], Дже-кобсон (Jacobson) [F 1985], Джамса (Jamsa) [F 1984], Кади (Kadie) [F 1986], Мазиеро и Германо (Masiero and Germano) [F 1988], Ниелсен (Nielsen) [F 1988], Ниес (Nies) [F 1986], Рэйлич и Сильва (Railich and Silva) [F 1987], Грэхем (Graham) [F 1987].

Сравнение различных процессов объектно-ориентированного развития можно найти в работах Арнольда (Arnold) [F 1991], Боем-Дэвиса и Росса (Boehm-Davis and Ross) [H 1984], деШампо (deChampeaux) [B 1991], Криббса, Муна и Ро (Cribbs, Moon, and Roe) [F 1992], Фоулер (Fowler) [F 1992], Келли (Kelly) [F 1986], Манино (Mannino) [F 1987], Сонга (Song) [F 1992], Вебстера (Webster) [F 1988]. Брукман (Brookman) [F 1991] и Фичмэн (Fichman) [F 1992] сравнили структурные и объектно-ориентированные методы.

Эмпирические исследования процессов создания программного обеспечения можно найти в работе Кертис (Curtis) [H 1992], а также в трудах Software Process Workshop [H 1988]. Еще одно интересное исследование принадлежит Гвиндону (Guindon) [H 1987], изучавшему процессы, которые разработчики использовали раньше. Речтин (Rechtin) [H 1992] предложил прагматическое руководство для системного архитектора, который должен управлять процессом развития.

Интересная ссылка по вопросу о "созревании" программного продукта - это работа Хэмфри (Hamphrey) [H 1989]. Классическая ссылка на то, как симитировать этот процесс, - статья Парнаса (Parnas) [H 1986].

Глава 7

Практические вопросы

Разработка программ пока остается чрезвычайно трудоемким делом, в значительной степени она по-прежнему больше напоминает строительство коттеджей, чем промышленное возведение зданий [1]. Доклад Кишиды и др. свидетельствует, что даже в Японии на начальной стадии проектов "все еще по большей части полагаются на неформальный подход - карандаш и бумагу" [2].
Ситуация усугубляется тем обстоятельством, что проектирование - никак не точная наука. Возьмем проектирование баз данных, одну из технологий, предшествовавших объектно-ориентированному проектированию. Как замечает Хаврис-кевич: "Хотя все выглядит просто и ясно, неизбежно примешивается изрядная доля личного представления о важности различных объектов на предприятии. В результате процесс проектирования не воспроизводим: разные проектировщики могут создать разные модели одного и того же предприятия" [3].
Из этого можно сделать вывод, что при любом самом изощренном и теоретически обоснованном методе проектирования нельзя игнорировать практические соображения. Значит, мы должны принять во внимание управленческий опыт в таких областях, как подбор кадров, управление релизами и контроль качества. Для технолога это в высшей степени скучная материя, но для разработчика это реалии жизни, с которыми надо справляться, чтобы создавать сложные программные системы. Итак, в этой главе мы займемся практическими вопросами объектно-ориентированной разработки и влиянием объектной модели на управление.

7.1.Управление и планирование

Если мы при проектировании опираемся на метод итеративного развития, то важнее всего иметь сильное руководство, способное управлять ходом проекта и направлять его. Слишком много проектов сбились с пути из-за неспособности сосредоточиться на главном, и только сильная команда менеджеров может что-то с этим поделать.

Управление риском

В конечном счете, главная обязанность менеджера программного продукта - управление как техническим, так и нетехническим риском. Технический риск для объектно-ориентированной системы содержится в решении таких проблем, как выбор структуры наследования классов, обеспечивающий наилучший компромисс между удобством и гибкостью программного продукта. Серьезное решение приходится также принимать при выборе механизмов упрощения архитектуры и улучшения эффективности. Нетехнический риск содержит в себе такие вопросы, как контроль своевременности поставки программных продуктов от третьих фирм или регулирование отношений заказчика и разработчиков, что необходимо для выяснения реальных требований к системе на стадии анализа.

Как было описано в предыдущей главе, микропроцесс объектно-ориентированной разработки нестабилен по своей природе и требует активного управления, концентрации усилий. К счастью, существует макропроцесс разработки, который выдвигает ряд конкретных требований и характеристик. Менеджер проекта, изучая соответствие требований и фактических результатов, может оценить состояние разработки и, при необходимости, перенаправить ресурсы команды. Эволюционная суть макропроцесса разработки означает, что можно распознать проблемы в начале

жизненного цикла и продуманно учесть связанный с ними риск прежде, чем проект окажется в опасности.

Многие виды деятельности по управлению разработкой программного обеспечения, например, планирование задач и просмотры, предусмотрены не только в объектно-ориентированной технологии. Однако при управлении объектно-ориентированным проектом намечаемые задачи и рассматриваемые результаты не совсем такие, как в других системах.

Планирование задач

Независимо от размера проекта, которым вы заняты, полезно раз в неделю проводить встречу всех разработчиков для обсуждения выполненной работы и действий на следующую неделю. Некоторая минимальная частота встреч необходима, чтобы способствовать общению между членами коллектива. С другой стороны, слишком частые встречи снижают продуктивность и обычно являются признаком потери курса. Объектно-ориентированная разработка требует, чтобы разработчики имели достаточное время для размышлений, введения новшеств и неформального общения с коллегами. Менеджеры команды должны учитывать в плане и это не структурированное время.

Проводимые встречи дают простую, но эффективную возможность гладкой подстройки планов в микропроцессе и распознания показавшихся на горизонте опасных ситуаций. Результатом такой встречи может быть небольшая корректировка в распределении работ, обеспечивающая устойчивость процесса: никакой проект не может позволить хотя бы одному из разработчиков сидеть сложа руки, ожидая, пока другие члены команды приведут в порядок свою часть архитектуры. Это особенно верно для объектно-ориентированных систем, в которых архитектура представляется набором классов и механизмов. Проект может заглохнуть, если разработчикам никак не удается разобраться с одним из ключевых классов.

Планирование задач связано с построением графика представления результатов макропроцесса. В промежутках между очередными релизами менеджеры команды должны оценить трудности, угрожающие проекту,[1] сконцентрировать ресурсы, чтобы разрешить возникшие проблемы, и далее заниматься новой итерацией микропроцесса, в результате которой нужно получить стабильную систему, удовлетворяющую сценариям, запланированным для нового релиза. Планирование задач на этом уровне очень часто оказывается неудачным из-за чрезмерно оптимистических графиков [4]. Разработка, которая рассматривалась как "просто вопрос программирования", растягивается на многие дни работы; графики выбрасываются в корзину, когда разработчик, занимаясь частью системы, предполагает определенные протоколы для других частей системы, а потом получает неполно или неправильно изготовленные классы. Смертельную опасность могут представлять внезапно обнаружившиеся ошибки в компиляторе или то, что программа не укладывается в заданное время исполнения. И то и другое часто приходится преодолевать, жертвуя принятыми ранее тактическими решениями.

Ключ к тому, чтобы не поддаваться чрезмерно оптимистическому планированию, - "калибровка" команды и ее инструментов разработки. Типичное планирование задач протекает следующим образом. Вначале менеджер направляет энергию разработчика на специфические части системы, например на проектирование классов для интерфейса с реляционной базой данных. Разработчик анализирует необходимые усилия и оценивает время исполнения, которое менеджер учитывает при планировании других его

---

[1] Глиб замечает: "если вы не идете в атаку на трудности, трудности идут в атаку на вас" [5].

действий. Проблема в том, что эти оценки не всегда реальны: они обычно делаются в расчете на самый благоприятный случай. Один разработчик может согласиться на решение задачи за неделю, а другой на эту же задачу попросит месяц. Когда работа будет реально выполнена, может оказаться, что она отняла три недели рабочего времени у обоих разработчиков: первый разработчик недооценил усилия (общая проблема многих программистов), а второй разработчик оценил реальные усилия более точно (например потому, что он понимал разницу между действительным рабочим временем и календарным, которое часто заполнено множеством нефункциональных действий). Таким образом, чтобы разработать графики, к которым коллектив может иметь доверие, менеджерам необходимо ввести своего рода "калибровочные коэффициенты" для пересчета оценок времени, заявленных разработчиками. Это не признак того, что менеджеры не доверяют разработчикам, но просто признание того факта, что большинство программистов сосредоточены на технических проблемах, а не на задачах планирования. Менеджер должен помогать разработчикам учиться планировать, - но это тот навык, который может быть приобретен только опытом.

Объектно-ориентированный процесс разработки помогает выявить явные принципы калибровки. Метод итеративного развития позволяет в начале проекта найти множество промежуточных пунктов, которые менеджеры команды использовали бы для накопления данных о достижениях каждого разработчика, определения графиков работы и планирования встреч. При эволюционной разработке руководители коллектива со временем будут лучше понимать реальную продуктивность каждого своего разработчика, а разработчики смогут научиться более точно оценивать объем предстоящей работы. Те же выводы приложимы и к инструментам: архитектурные релизы уже на ранней стадии проекта стимулируют использование инструментов разработки, которые помогают своевременно проверить структурные ограничения.

Просмотр

Просмотр (walkthroughs) - общепринятая практика, которую нужно использовать каждой команде разработчиков. Как и планирование задач, просмотр программного обеспечения был введен независимо от объектно-ориентированной технологии. Однако при просмотре не объектно-ориентированных систем внимание обращается на другое.

Руководитель должен проводить просмотры с разумной частотой. За исключением самых ответственных и уязвимых для ошибок мест, просто неэкономично проверять каждую строчку программы. Следовательно, руководитель должен направить ограниченные ресурсы своей команды на рассмотрение проблем, опасных для стратегии разработки. Для объектно-ориентированных систем это означает большую формальность при проведении просмотров сценариев и архитектуры системы и менее формальную проверку тактических решений.

Как описано в предыдущей главе, сценарии являются первичным результатом объектно-ориентированного анализа. Они должны выражать требуемое поведение системы в терминах ее функциональных точек. Формальные просмотры сценариев проводятся аналитиками команды, вместе с экспертами предметной области или конечными пользователями при возможном участии других разработчиков. Лучше проводить такие просмотры на протяжении всей стадии анализа, чем ожидать выполнения одного глобального просмотра по завершении анализа, когда будет уже слишком поздно сделать что-нибудь полезное, перенаправив усилия аналитиков. Эксперименты показывают, что даже непрограммисты могут понять сценарии,

представленные в виде текста или диаграмм объектов.[2] В конечном счете просмотр помогает выработать общий словарь для разработчиков и пользователей системы. Привлечение к участию в просмотре других членов команды способствует уяснению ими реальных требований к системе на ранних этапах разработки.

Просмотр архитектуры должен охватывать всю систему, включая ее механизмы и структуру классов. Как и при просмотре сценариев, просмотр архитектуры (архитектором или другими проектировщиками) должен производиться на протяжении всего проекта. Сначала просмотр сосредоточен на общих архитектурных решениях, а позднее, возможно, он акцентируется на некоторых категориях классов или конкретных механизмах. Основная цель просмотра состоит в проверке архитектуры в начале жизненного цикла и выработке общего взгляда на нее. Вторичной целью является поиск повторяющихся шаблонов классов или взаимодействий, которые затем могут быть использованы для упрощения архитектуры.

Неформальный просмотр следует проводить еженедельно. На нем обычно рассматриваются некоторые группы классов или механизмы нижнего уровня. Цель - проверить тактические решения; побочная цель - дать возможность старшим разработчикам научить новичков.

7.2. Кадры

Распределение ресурсов

Один из наиболее замечательных аспектов управления объектно-ориентированными проектами - это тот факт, что в устойчивом состоянии обычно наблюдается сокращение необходимых ресурсов и изменяется график их расходования по сравнению с традиционными методами. Именно "в устойчивом состоянии". Вообще говоря, первый объектно-ориентированный проект, предпринятый организацией, потребует несколько больше ресурсов - главным образом, в соответствии с кривой обучения, описывающей адаптацию ко всякой новой технологии. Выгоды проявятся во втором или третьем проекте, когда разработчики наберутся опыта в проектировании классов, поиске общих абстракций и механизмов, а менеджеры освоятся с методом итеративного развития.

На стадии анализа потребность в ресурсах с переходом на объектно-ориентированные методы обычно мало изменяется. Однако, поскольку объектно-ориентированный процесс уделяет больше внимания архитектуре, мы стремимся привлекать архитекторов и других разработчиков как можно раньше, иногда начиная архитектурные эксперименты еще на последней стадии анализа. Во время эволюции, как правило, потребуется меньше ресурсов, потому что работа облегчится общими абстракциями и механизмами, изобретенными ранее при проектировании архитектуры или выпуске предварительных версий. Тестирование может также потребовать меньше ресурсов, потому что новые функции обычно добавляются к уже корректно ведущей себя структуре класса или механизму. Таким образом, тестирование начинается раньше и является скорее постоянным и постепенным, чем разовым действием. Интеграция обычно требует значительно меньших ресурсов по сравнению с традиционными методами, главным образом потому, что она тоже происходит постепенно, от релиза к релизу, а не одним броском. Таким образом, в устойчивом состоянии трудозатраты оказываются гораздо меньше, чем при традиционных подходах. Более того, если учесть эксплуатационные затраты, то окажется, что весь жизненный цикл объектно-ориентированных программ часто стоит дешевле,

---

[2] Мы встречались с использованием этой системы обозначении в работе таких непрограммистских групп как астрономы, биологи, метеорологи, физики и банкиры.

так как конечный продукт, скорее всего, будет лучшего качества и окажется более приспособленным к изменениям.

Роли разработчиков

Полезно помнить, что разработка программного продукта в конечном счете производится людьми. Разработчики - не взаимозаменяемые части, и успешное создание любой сложной системы требует уникальных и разнообразных навыков всех членов целеустремленного коллектива.

Эксперименты показывают, что объектно-ориентированная разработка требует несколько иного разделения труда по сравнению с традиционными методами. Мы считаем следующие три роли разработчиков важнейшими в объектно-ориентированном подходе:

- архитектор проекта
- ответственные за подсистемы
- прикладные программисты.

Архитектор проекта - его творец, человек с сильно развитым воображением; он отвечает за эволюцию и сопровождение архитектуры системы. Для малых или средних систем архитектурное проектирование обычно выполняется одной, максимум двумя светлыми личностями. Для больших проектов эта обязанность может быть распределена в большом коллективе. Архитектор проекта - не обязательно самый главный разработчик, но непременно такой, который может квалифицированно принимать стратегические решения (как правило благодаря обширному опыту в построении систем такого типа). Благодаря опыту, разработчики интуитивно знают, какие общие архитектурные шаблоны уместны в данной предметной области и какие проблемы эффективности встают в определенных архитектурных вариантах. Архитекторы - не обязательно лучшие программисты, хотя они должны уметь программировать. Точно так же, как строительные архитекторы должны разбираться в строительстве, неблагоразумно нанимать архитектора программного обеспечения, который не является приличным программистом. Архитекторы проекта должны также быть сведущи в обозначениях и организации процесса объектно-ориентированной разработки, потому что они должны в конечном счете выразить свое архитектурное видение в терминах кластеров классов и взаимодействующих объектов.

Очень плохая практика - нанимать архитектора со стороны, который, образно выражаясь, въезжает на белом коне, провозглашает архитектурные принципы, а потом уматывает куда-то, в то время как другие пытаются справиться с последствиями его решений. Гораздо лучше привлечь архитектора к активной работе уже при проведении анализа и оставить его на как можно более длительный срок, даже на все время эволюции системы. Тогда он освоится с действительными потребностями системы и со временем испытает на себе последствия своих решений. Кроме того, сохраняя в руках одного человека или небольшой команды разработчиков ответственность за архитектурную целостность, мы повышаем шансы получить гибкую и простую архитектуру.

Ответственные за подсистемы - главные творцы абстракций проекта. Они отвечают за проектирование целых категорий классов или подсистем. Каждый ответственный в сотрудничестве с архитектором проекта разрабатывает, обосновывает и согласует с другими разработчиками интерфейс своей категории классов или подсистемы, а потом возглавляет ее реализацию, тестирование и выпуск релизов в течение всей эволюции системы.

Ответственные за подсистемы должны хорошо знать систему обозначений и организацию процесса объектно-ориентированной разработки. Обычно они программируют лучше чем архитекторы проекта, но не

располагают обширным опытом последних. Лидеры подсистем составляют от трети до половины численности команды.

Прикладные программисты (инженеры) - младшие по рангу участники проекта. На них возложено выполнение двух обязанностей. Некоторые из них отвечают за реализацию категории или подсистемы под руководством ее ведущего. Эта деятельность может включать в себя проектирование некоторых классов, но в основном связана с реализацией и последующим тестированием классов и механизмов, разработанных проектировщиками команды. Другие отвечают за написание классов, спроектированных архитектором и ответственными за подсистемы, реализуя тем самым функциональные точки системы. В некотором смысле, эти программисты занимаются написанием маленьких программ на языке предметной области, определенном классами и механизмами архитектуры.

Инженеры разбираются в системе обозначений и в организации процесса разработки, но не слишком блестяще; зато они, как правило, являются очень хорошими программистами, знающими основные идиомы и слабые места выбранных языков программирования. Инженеры составляют половину команды или более того.

Разница в квалификации ставит проблему подбора кадров перед всеми организациями, которые обычно имеют несколько сильных проектировщиков и большее количество менее квалифицированного персонала. Социальная польза нашего подхода к кадровой политике состоит в том, что он открывает путь для карьеры начинающим сотрудникам: молодые разработчики работают под руководством более опытных. Когда они наберутся опыта в использовании хорошо определенных классов, они смогут сами проектировать классы. Вывод: не обязательно каждому разработчику быть экспертом по абстракциям, но каждый разработчик может со временем этому научиться.

В больших проектах могут потребоваться и другие роли. Большинство из них (например, роль специалиста в средствах разработки) явно не связаны с объектно-ориентированной технологией, но некоторые непосредственно вытекают из нее (такие, как инженер, отвечающей за повторное использование):

• Менеджер проекта
Отвечает за управление материалами проекта, заданиями, ресурсами и графиком работ.

• Аналитик

Отвечает за развитие и интерпретацию требований конечных пользователей; должен быть экспертом в проблемной области, однако его не следует изолировать от остальной команды разработчиков.

• Инженер по повторному использованию

Управляет хранилищем (репозитарием) материалов проекта; участвуя в просмотре и других действиях, активно ищет общее и добивается его использования; находит, разрабатывает или приспосабливает компоненты для общего использования в рамках конкретного проекта или целой организации.

• Контролер качества

Измеряет результаты процесса разработки; задает общее направление (на системном уровне) тестирования всех прототипов и релизов.

• Менеджер интеграции

Отвечает за сборку совместимых друг с другом версий категорий и подсистем в релизы; следит за их конфигурированием.

• Инструментальщик

Отвечает за создание и адаптацию инструментов программирования, которые облегчают производство программ и (особенно) генерацию кода.
Библиотека должна содержать семейство классов, объединенных согласованным внешним интерфейсом, но с разными представлениями, так чтобы разработчики могли выбрать то, семантика которого наиболее точно соответствует приложению.Предполагает тестирование отдельных классов и механизмов; является обязанностью инженера, который их реализовал.

Тестирование должно фокусироваться на внешнем поведении системы; его побочная цель - определить границы системы чтобы понять, как она может выходить из строя при определенных условиях.

7.4. Повторное использование

Элементы повторного использования
Любой программный продукт (текст программы, архитектура, сценарий или документация) может быть использован повторно. Как сказано в главе 3, в объектно-ориентированных языках программирования первичным лингвистическим средством повторного использования являются классы: класс может порождать подклассы, специализирующие или дополняющие его. Далее, в главе 4 говорилось о повторном использовании шаблонов классов, объектов и элементов проектирования в форме идиом, механизмов и сред разработки. Повторное использование шаблонов находится на более высоком уровне абстракции по сравнению с использованием индивидуальных классов и дает больший выигрыш (хотя оно труднее достижимо).

Не следует доверять цифрам, характеризующим повторное использование [9]. В удачных проектах, с которыми мы сталкивались, количество повторно использованных элементов доходило до 70% (то есть почти три четверти программного обеспечения системы было взято без изменений из некоторого другого источника), но бывало и нулевым. Не следует думать, что повторное использование должно достичь некоторой обязательной величины; возможность повторного использования сильно зависит от предметной области и нетехнических факторов, таких, например, как степень напряженности рабочего графика, природа отношений с субподрядчиками и соображения безопасности.

Безусловно, любой процент повторного использования лучше, чем нулевой, так как экономит ресурсы, которые иначе пришлось бы потратить еще раз.

Библиотека должна быть относительно небольших размеров; надо всегда помнить, что пользователь с большей охотой займется разработкой собственного кода, чем изучением чужого малопонятного класса.

Предполагается наличие трансляторов языка C++, поддерживающих параметризованные классы и обработку исключений. В целях обеспечения переносимости библиотеки она не должна зависеть от служб операционной системы.

Таким образом, первым результатом нашего анализа будет разделение всех абстракций на две категории:

- Структуры — Содержит все структурные абстракции
- Инструменты — Содержит все алгоритмические абстракции

Как мы скоро увидим, между этими двумя категориями существует отношение использования: некоторые инструменты построены на базе более примитивных свойств, обеспечиваемых структурами.

На втором этапе анализа мы постараемся выделить базовые классы, которые могут быть использованы в различных стандартных программах (чем шире будет круг рассмотренных приложений, тем лучше). Если в результате окажется, что некоторые из данных классов имеют много общего с абстракциями, определенными на первой стадии анализа, это будет знаком того, что ключевые абстракции были выявлены правильно. Можно составить длинный список специфических абстракций, присущих конкретным видам человеческой деятельности: валюта, астрономические координаты, единицы измерения массы и длины. Мы не будем включать подобные абстракции в нашу библиотеку, так как они либо слишком плохо поддаются формализации (валюта), либо очень специфичны (астрономические координаты), либо настолько примитивны, что нет смысла организовывать специально для них отдельные классы (единицы измерения массы и длины).

Проведя анализ, мы выделим следующие типы структур:

- Набор — Множество различных элементов (в том числе дубликатов).
- Множество — Набор неповторяющихся элементов.
- Коллекция — Индексируемое множество элементов.
- Список — Последовательность элементов, имеющая начало; структурное разделение допускается.
- Стек — Последовательность элементов; элементы могут удаляться и добавляться только с одного конца.
- Очередь — Последовательность элементов, к которой можно добавлять элементы с одного конца, а удалять - с другого.
- Дека — Последовательность элементов, к которой можно добавлять и из которой можно удалять элементы с обоих концов.
- Кольцо — Последовательность элементов, к которой можно добавлять и из которой можно удалять элементы, находящиеся на вершине круговой структуры.
- Строка — Индексируемая последовательность элементов, в которой возможны операции с подстроками.
- Ассоциативный Словарь пар "элемент/значение". массив
- Дерево — Набор (имеющий начало - корень дерева) вершин и ребер, которые не могут образовывать циклы и пересекаться; структурное разделение допускается.
- Граф — Множество вершин и ребер (без выделенного начального элемента), которое может содержать циклы и пересечения;

структурное разделение допускается.

Как уже говорилось в главе 4, упорядочение представленных выше абстракций есть проблема классификации. Мы выбрали именно такую модель из-за того, что она обеспечивает закрепление определенного поведения за каждой категорией объектов.

Обратите внимание на типы поведения, которые использовались в качестве критериев при разбиении на классы: некоторые структуры ведут себя как коллекции (наборы и множества), а другие - как последовательности (деки и стеки). В некоторых структурах (графы, списки и деревья) возможно структурное разделение, в то время как остальные более монолитны и не допускают структурного разделения своих элементов. Как мы увидим далее, подобная классификация поможет в дальнейшем сформировать достаточно простую архитектуру системы.

Для некоторых классов в процессе анализа выявилась желательность их функциональной изменчивости. В частности, нам могут понадобиться упорядоченные коллекции, деки и очереди (последние часто называют *приоритетными очередями*[19]). Кроме того, мы можем различать ориентированные и неориентированные графы, односвязные и двусвязные списки, бинарные, множественные и AVL-деревья [20]. Эти специализированные абстракции могут быть получены уточнением одной из вышеперечисленных; их не следует выделять в отдельные большие категории.

Несмотря на то, что мы уже обнаружили признаки общности поведения, мы пока не будем заниматься проработкой иерархической структуры. На этапе анализа важно разобраться в ролях каждой абстракции.

Мы выделим следующие типы инструментов:

• Дата/Время Операции с датой и временем.

• Фильтры Ввод, обработка и вывод.

• Поиск по образцу Операции поиска последовательностей внутри других последовательностей

• Поиск Операции поиска элементов внутри структур

• Сортировка Операции упорядочивания структур

• Утилиты Составные операции, базирующиеся на базовых структурных операциях.

Несомненно, существует масса различных функциональных вариантов этих абстракций. Можно, например, выделить несколько видов сортировок (быстрая сортировка методом пузырька, сортировка кучи и т. д.) или поиска (последовательный, двоичный, различные способы обхода дерева и т. д.). Как и раньше, мы отложим решения относительно наследования этих абстракций.

Модели взаимодействий

Итак, мы определили основные функциональные элементы нашей библиотеки;

однако изолированные абстракции сами по себе - еще не среда разработки. Как отметил Вирфс-Брок: "Среда разработки предоставляет пользователю модель взаимодействий между объектами входящих в нее классов... Чтобы освоить среду разработки, прежде всего следует изучить методы взаимодействия и ответственности ее классов". Это и есть тот критерий, по которому можно отличить среду разработки от простого набора классов: среда - это совокупность классов и механизмов взаимодействия экземпляров этих классов.

Анализ показывает, что существует определенный набор основных механизмов, необходимый для библиотеки базовых классов:

• семантика времени и памяти

• управление хранением данных

• обработка исключений

• идиомы итерации

• синхронизация при многопоточности.

При проектировании системы базовых классов необходимо сохранять баланс между перечисленными техническими требованиями.[21] Если мы будем пытаться решить каждую задачу по отдельности, то, скорее всего, получим ряд изолированных решений, не связанных между собой ни общими протоколами, ни общей концепцией, ни реализацией. Такой наивный подход приведет к изобилию различных подходов, которое испугает потенциального пользователя получившейся библиотеки.

Встанем на точку зрения пользователя нашей библиотеки. Какие абстракции представляют имеющиеся в ней классы? Как они взаимодействуют между собой? Как их можно приспособить к предметной области? Какие классы играют ключевую роль, а какие можно вообще не использовать? Вот те вопросы, на которые нужно дать ответ перед тем, как предлагать пользователям библиотеку для решения нетривиальных задач. К счастью для пользователя, ему не обязательно во всех деталях представлять себе, как работает библиотека, подобно тому, как не нужно понимать принципы работы микропроцессора для программирования на языке высокого уровня. В обоих случаях реализации нижнего уровня может быть продемонстрирована каждому пользователю, но только при его желании.

Рассмотрим описание абстракций нашей библиотеки с двух точек зрения:

пользователя, который только объявляет объекты уже существующих классов, и клиента, который конструирует собственные подклассы на базе библиотечных. При проектировании с расчетом на первого пользователя желательно как можно сильнее ограничить доступ к реализациям абстракций и сконцентрироваться на их ответственностях; проектирование с учетом запросов второго пользователя предполагает открытость некоторых внутренних деталей реализации, однако, не настолько, чтобы стало возможным нарушить фундаментальную семантику абстракции. Таким образом, приходится отметить некоторую противоречивость основных требований к системе.

Одной из главных проблем при работе с большой библиотекой являются трудности в понимании того, какие, собственно, механизмы она включает в себя. Перечисленные выше модели представляют собой как бы душу архитектуры библиотеки:

чем больше разработчик знает об этих механизмах, тем легче ему будет использовать существующие в библиотеке компоненты, а не сочинять с нуля собственные. На практике получается так, что пользователь сначала знакомится с содержанием и работой наиболее простых классов, и только затем, проверив надежность их работы, постепенно начинает использовать все более сложные классы. В процессе разработки, по мере того как начинают вырисовываться новые, присущие предметной области пользователя, абстракции, они тоже могут добавляться в библиотеку. Развитие объектно-ориентированной библиотеки - это длительный процесс, проходящий через ряд промежуточных этапов.

Именно так мы будем строить нашу библиотеку: сначала определим тот архитектурный минимум, который реализует все пять выделенных нами механизмов, и затем начнем постепенно наращивать на этом остове все новые и новые функции.

9.2. Проектирование

Тактические вопросы

В соответствии с законом разработки программ Коггинса "прагматизм всегда должен быть предпочтительней элегантности, ведь Природу все равно ничем не удивить". Следствие: проектирование никогда не будет полностью независимым от языка реализации проекта. Особенности языка неизбежно наложат отпечаток на те или иные архитектурные решения, и их игнорирование может привести к тому, что нам придется работать в дальнейшем с абстракциями, не в полной мере учитывающими преимущества и недостатки конкретного языка реализации.

Как было отмечено в главе 3, объектно-ориентированные языки предоставляют три основных механизма упорядочения большего числа классов: наследование, агрегацию и параметризацию. Наследование является наиболее популярным свойством объектно-ориентированной технологии, однако далеко не единственным принципом структурирования. Как мы увидим, сочетание параметризации с наследованием и агрегацией помогает создать достаточно мощную и в то же время компактную архитектуру.

Рассмотрим усеченное описание предметно-зависимого класса очереди в C++:

```
class NetworkEvent... // сетевое событие

class EventQueue {// очередь событий
public:
	EventQueue();
	virtual ~EventQueue();

	virtual void clear(); // очистить
	virtual void add(const NetworkEvent&); // добавить
	virtual void pop(); // продвинуть

	virtual const NetworkEvent& front() const; // первый
элемент
	...
};
```

Перед нами абстракция, олицетворяющая очередь событий: структура, в которую мы можем добавлять новые элементы в конец очереди и удалять элементы из начала очереди. C++ позволяет скрыть внутренние детали реализации класса очереди за его внешним интерфейсом (операциями **clear** , **add**, **pop** и **front**).

Нам могут потребоваться также некоторые другие варианты очереди, например, приоритетная очередь, где события выстраиваются в соответствии с их срочностью. Разумно воспользоваться результатами уже проделанной работы и организовать новый класс на базе ранее определенного:

```
class PriorityEventQueue : public EventQueue {
public:
	PriorityEventQueue();
	virtual ~PriorityEventQueue();

	virtual void add(const NetworkEvent&);
	…
};
```

Виртуальность функций (например функции **add**) поощряет переопределение операций в подклассах.

Комбинация наследования с параметризованными классами позволяет создавать еще более общие абстракции. Семантика класса очереди не зависит от того, что в ней: волки или овцы. Используя классы-шаблоны, можно переопределить наш базовый класс следующим образом:

```
template<class Item>
```

```
class Queue {
public:
      Queue ();
      virtual ~Queue() ;
      virtual void clear();
      virtual void add(const Item&);
      virtual void pop();
      virtual const Item& front() const;
      …
};
```

Это наиболее распространенный способ использования параметризованных классов: взять существующий конкретный класс, выделить в нем то, что не зависит от элементов, с которыми он оперирует, и сделать эти элементы аргументами шаблона.

Наследование и параметризация очень хорошо сочетаются. Наш подкласс **PriorityQueue** можно, например, обобщить следующим образом:

```
template<class Item>
class PriorityQueue : public Queue<Item> {
public:
      PriorityQueue();
      virtual ~PriorityQueue();

      virtual void add(const Item&);
      …
};
```

Безопасность с точки зрения типов - ключевое преимущество данного подхода. Мы можем создать целый ряд различных классов конкретных очередей:

```
      Queue<char> characterQueue;
      typedef Queue<MetworkEvent> EventQueue;
      typedef PriorityQueue<NetworkEvent> PriorityEventQueue;
```

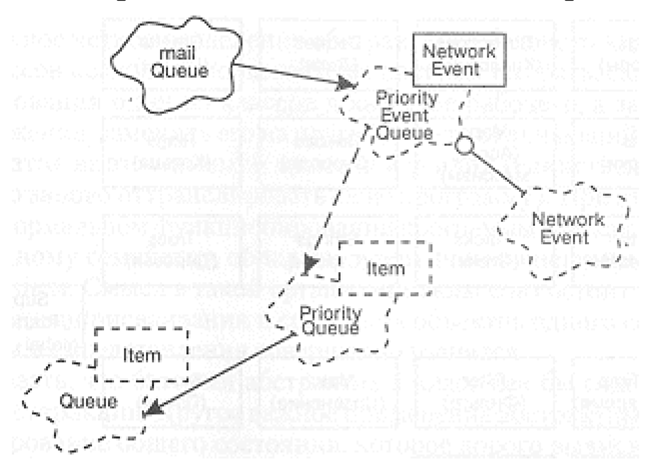


Рис. 9-1. Наследование и параметризация

При этом язык реализации не позволит нам присоединить событие к очереди символов, а вещественное число - к очереди событий.

Рис. 9-1 иллюстрирует отношения между параметризованным классом (**Queue**), его подклассом (**PriorityQueue**), примером этого подкласса (**PriorityEventQueue**) и одним из его экземпляров (**mailQueue**).

Этот пример подтверждает правильность одного из самых первых наших архитектурных решений: почти все классы нашей библиотеки должны быть параметризованными. Тогда будет выполнено и требование защищенности.

Макроорганизация

Как уже отмечалось в предыдущих главах, классы есть необходимое, но не достаточное средство декомпозиции системы. Это замечание в полной мере касается и библиотеки классов. Неупорядоченный набор классов, в котором разработчики копаются в поисках чего-либо полезного, - едва ли не худшее из возможных решений. Лучше разбить классы на отдельные категории (рис. 9-2). Такое решение позволяет удовлетворить требованию простоты библиотеки.

При первом взгляде на проблемную область легко заметить, что мы могли бы воспользоваться общими функциональными свойствами классов. Поэтому заведем общедоступную категорию `Support` (поддержка) для абстракций низкого уровня и классов, поддерживающих общие механизмы библиотеки.

Это наблюдение приводит нас ко второму принципу архитектуры библиотеки: четкое разделение между политикой и реализацией. Такие абстракции, как очереди, множества и кольца, отражают политику использования низкоуровневых структур: связных списков или массивов. Очередь, например, выражает политику, при которой можно только удалять элементы из начала структуры и добавлять элементы к ее концу. Множество, с другой стороны, не представляет никакой политики, требующей упорядочения элементов. Кольцо требует упорядочения, но предполагает, что начальный и конечный элемент соединены. К категории `Support` мы будем относить простые абстракции - те, над которыми надстраивается политика.

Поместив эту категорию классов в код библиотеки, мы поддерживаем библиотечное требование расширяемости. Основная масса разработчиков, может быть, и не будет использовать классы из `Support`. Однако разработчики библиотек и более

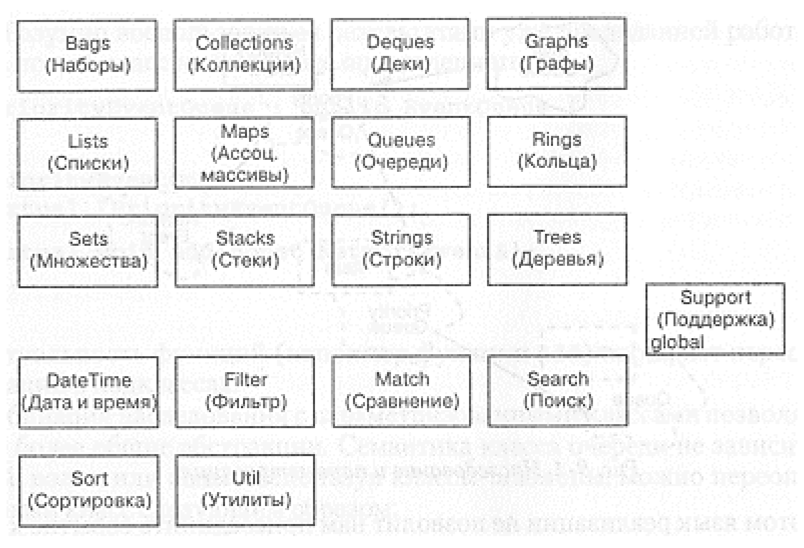


Рис. 9-2. Категории классов в библиотеке

продвинутые программисты смогут задействовать базовые абстракции из `Support` для конструирования новых классов или модификации поведения существующих.

Как видно из рис. 9-2, библиотека организована не в виде дерева, а в виде леса классов; здесь не существует единого базового класса, как этого требуют языки типа Smalltalk.

На рисунке этого не видно, но на самом деле классы категорий `Graphs`, `Lists` и `Trees` несколько отличаются от других структурных классов. Еще раньше мы отмечали, что абстракции типа деки и стека являются монолитными. С монолитной структурой можно иметь дело только как с единым целым: ее нельзя разбить на отдельные идентифицируемые компоненты, и таким образом гарантируется ссылочная целостность. С другой

стороны, в композитной структуре (такой как граф) структурное разделение допускается. В ней мы можем, например, получать доступ к подспискам, ветвям дерева, отдельным вершинам или ребрам графа. Фундаментальное различие между этими двумя категориями структур лежит в семантике операций копирования, присваивания и сравнения. Для монолитных абстракций подобные операции можно назвать "глубокими", а для композитных абстракций - "поверхностными", в том смысле, что при копировании происходит передача ссылки на часть общей структуры.

Семейства классов

Третий основной принцип проектирования библиотеки заключается в построении семейств классов, связанных отношением наследования. Для каждого типа структур мы создадим несколько различных классов, объединенных единым интерфейсом (как в случае с абстрактным классом `Queue`), но с разными конкретными подклассами, имеющими несколько различные представления и поэтому отличающимися Своим устройством и характеристиками "время/память". Таким образом мы обеспечим библиотечное требование полноты. Разработчик сможет выбрать тот конкретный класс, который в большей степени подходит для решения его задачи. В то же время этот класс обладает тем же интерфейсом, что и остальные классы семейства. Сознательное четкое разделение абстрактного базового класса и его конкретных подклассов позволяет пользователю системы выбрать, скажем, на первом этапе проектирования один из классов в качестве рабочего, а затем, в процессе доводки приложения, заменить его на другой, чем-то отличающийся класс того же семейства, затратив на это минимум времени и усилий (единственное, что ему потребуется, - это заново оттранслировать свою программу). При этом разработчик будет уверен в нормальном функционировании программы, так как все классы, принадлежащие одному семейству, обладают идентичным внешним интерфейсом и схожим поведением. Смысл в такой организации классов состоит еще и в возможности копирования, присваивания и сравнения объектов одного семейства даже в том случае, если их представления совершенно разнятся.

Можно сказать, что базовый абстрактный класс как бы содержит в себе все важные черты абстракции. Другое важное применение абстрактных базовых классов - это кэширование общего состояния, которое дорого вычислять заново. Так можно перевести вычисление $O(n)$ в операцию порядка $O(1)$ - простое считывание данных. При этом, естественно, требуется обеспечить соответствующий механизм взаимодействия между абстрактным базовым классом и его подклассами, чтобы гарантировать актуальность кэшируемого значения.

Элементы семейства классов представляют собой различные *формы* абстракции. Опыт показывает, что существуют две основные формы абстракций, которыми следует пользоваться разработчику при создании серьезных приложений. Во-первых, это форма конкретного представления абстракции в оперативной памяти машины. Существует два варианта такого представления: выделение памяти для структуры из стека или выделение оперативной памяти из кучи. Им соответствуют две формы абстракций: *ограниченная* и *неограниченная*:

• Ограниченная Структура хранится в стеке и, таким образом, имеет статический размер (известный в момент создания объекта).

• Неограниченная Структура хранится в куче и ее размеры могут динамически изменяться.

Так как ограниченная и неограниченная формы абстракции имеют общие интерфейс и поведение, их обе можно представить в виде прямых

подклассов абстрактного базового класса для каждой структуры. Мы обсудим эти и другие особенности организации данных в следующих разделах.

Второй вариант связан с синхронизацией. Как было отмечено в главе 2, множество полезных приложений обходятся одним процессом. Их называют *последовательными* системами, потому что они используют один поток управления. Для других приложений (особенно это касается систем реального времени) требуется обеспечить синхронизацию нескольких одновременно выполняемых потоков. Такие системы называются *параллельными*, и в них каким-то образом должно обеспечиваться взаимное исключение процессов, конкурирующих за один и тот же ресурс. Ясно, что нельзя дать возможность управлять одним и тем же объектом одновременно нескольким потокам, это в конце концов приведет к нарушению его состояния. Рассмотрим, например, поведение двух агентов, которые одновременно пытаются добавить элемент одному и тому же объекту класса `Queue`. Первый агент, начавший добавление элемента, может быть прерван раньше, чем окончит данную операцию, и оставит объект второму агенту в незавершенном состоянии.

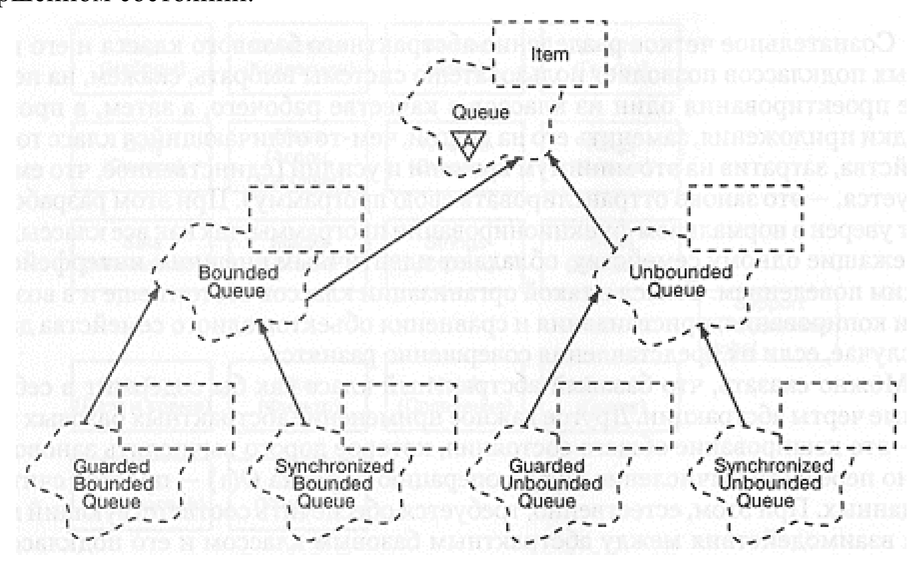


Рис. 9-3. Семейства классов

Как отмечалось в главе 3, в данном случае при проектировании существуют всего три возможных альтернативы, каждая из которых требует обеспечения различного уровня взаимодействия между агентами, оперирующими с общими объектами:

• последовательный

• защищенный

• синхронизированный.

Мы рассмотрим каждый из этих вариантов более подробно в следующем разделе. Обеспечение взаимодействия между абстрактным базовым классом, формами его представления и формами синхронизации порождает для каждой структуры семейство классов, подобное тому, которое приведено на рис. 9-3. Теперь можно понять, почему мы в свое время решили организовать библиотеку именно в виде семейств классов, а не в виде единого дерева. Это было сделано из-за того, что такая архитектура:

• Отражает общность различных форм.

• Позволяет осуществлять более простой доступ к элементам библиотеки.

• Позволяет избежать бесконечных метафизических споров о "чистом объектно-ориентированном подходе".

• Упрощает интеграцию системы с другими библиотеками.

Микроорганизация

В целях обеспечения простоты работы с системой выберем один общий стиль оформления структур и механизмов библиотеки:

```
template<...>
class Name : public Superclass {
	public:
	// конструкторы // виртуальный деструктор
	// операторы
	// модификаторы
	// селекторы
protected:
	// данные
	// служебные функции
private:
	// Друзья
};
```

Описание абстрактного базового класса **Queue** начинается следующим образом:

```
template<class Item> class Queue {
```

Сигнатура шаблона **template** служит для задания аргументов параметризованного класса. Отметим, что в C++ шаблоны сознательно введены таким образом, чтобы передать достаточную гибкость (и ответственность) в руки разработчика, инстанцирующего шаблон в своем приложении.

Далее определим обычный список конструкторов и деструкторов:

```
Queue();
Queue(const Queue<Item>&);
virtual -Queue() ;
```

Отметим, что мы описали деструктор виртуальным, чтобы обеспечить полиморфное поведение при уничтожении объектов класса. Далее объявим все операторы:

```
virtual Queue<Item>& operator=(const Queue<ltem>&) ;
virtual int operator==(const Queue<Item>&) const;
int operator!=(const Queue<Item>&) const;
```

Мы определили оператор присваивания (**operator==**) и оператор сравнения (**operator==**) как виртуальные для того, чтобы обеспечить безопасность типов. Переопределение этих операторов входит в обязанности подклассов. В них будут использоваться функции, аргументом которых является объект собственного специализированного класса. В этом смысле подклассы имеют то преимущество, что они знают представление своих экземпляров и могут обеспечить очень эффективную реализацию. Когда конкретный подкласс очереди неизвестен (например, если мы передаем объект по ссылке на его базовый класс), вызывается оператор базового класса, использующий может быть менее эффективные, но более универсальные алгоритмы. Эта идиома имеет побочный эффект: возможность работы одной и той же функции с очередями, имеющими различную внутреннюю реализацию, без нарушения типизации.

Если мы хотим ограничить доступ к копированию, присваиванию или сравнению некоторых объектов, нам надо объявить эти операторы защищенными или закрытыми.

Определим теперь модификаторы, позволяющие менять состояние объекта:

```
virtual void clear() = 0;
virtual void append(const Item&) = 0;
virtual void pop() =0;
```

```
virtual void remove (unsigned int at) = 0;
```

Данные операции объявлены как чисто виртуальные, а это значит, что их описание является обязанностью подклассов. Наличие чисто виртуальных функций делает класс **Queue** абстрактным.

Спецификатор **const** указывает (компилятор может это проверить) на использование функций-селекторов, то есть функций, предназначенных исключительно для получения информации о состоянии объекта, но не для изменения состояния:

```
virtual unsigned int length() const = 0;
virtual int isEmpty() const = 0;
virtual const Item& front() const =0;
virtual int location(const Item&) const = 0;
```

Эти операции тоже определены как чисто виртуальные, потому что класс **Queue** не обладает достаточной информацией для их полного описания.

Защищенная часть каждого класса начинается с описания тех элементов, которые формируют основу его структуры и должны быть доступны подклассам.[22] Абстрактный класс **Queue**, в. отличие от своих подклассов (см. ниже), подобных элементов не имеет.

Продолжит защищенную часть базового класса определение служебных функций, которые будут полиморфно реализованы в конкретных подклассах. Класс **Queue** содержит довольно типичный список таких функций:

```
virtual void purge () =0;
virtual void add(const Item&) = 0;
virtual unsigned int cardinality() const = 0;
virtual const Item& itemAt (unsigned int) const =0;
virtual void lock();
virtual void unlock();
```

Причины, по которым мы ввели именно эти функции, будут рассмотрены в следующем разделе.

И, наконец, определим закрытую часть, обычно содержащую объявления о классах-друзьях и те элементы, которые мы хотим сделать недоступными даже для подклассов. Класс **Queue** содержит только декларации о друзьях:

```
friend class QueueActiveIterator<Item>;
friend class Queuepassivelterator<ltem>;
```

Как мы увидим в дальнейшем, эти объявления друзей понадобятся для поддержки идиом итератора.

Семантика времени и памяти

Из пяти основных принципов строения библиотеки базовых классов, возможно, наиболее важен механизм, обеспечивающий клиента альтернативной простанственно-временной семантикой внутри каждого семейства классов.

Рассмотрим тот спектр требований, который должен учитываться при разработке библиотеки общего назначения. На рабочей станции, обладающей большим виртуальным адресным пространством, пользователь скорее всего будет расточать память ради более высокого быстродействия. С другой стороны, в некоторых встроенных системах, таких, как спутник или автомобильный мотор, ресурсы памяти часто ограничены, и разработчик вынужден выбирать в качестве рабочих те абстракции, которые используют меньше памяти (например, выделяя место под данные в стеке, а не в "куче"). Ранее мы различили эти две возможности как ограниченную и неограниченную формы соответственно.

Неограниченные формы применимы в тех случаях, когда размер структуры не может быть предсказан, а выделение и утилизация памяти из кучи не приводит ни к потере времени, ни к снижению надежности (как это бывает в некоторых приложениях, критичных по времени).[23] Ограниченные формы лучше подходят для работы с небольшими структурами, размер которых достаточно хорошо предсказуем. Учтем также, что динамическое выделение памяти менее терпимо к ошибкам программиста.

Таким образом, все структуры данной библиотеки должны присутствовать в альтернативных вариантах; поэтому нам придется создать два низкоуровневых класса поддержки, **Unbounded** (неограниченный) и **Bounded** (ограниченный). Задачей класса **unbounded** является поддержка быстро работающего связного списка, элементы которого размещаются в памяти, выделенной из "кучи". Это представление эффективно по скорости, но не по памяти, так как каждый элемент списка должен, кроме своего значения, дополнительно содержать указатель на следующий элемент того же списка. Задача класса **Bounded** состоит в организации структур на базе массива, что эффективно с точки зрения памяти, но добиться большой производительности трудно, так как, например, при добавлении элемента в середину списка приходится последовательно копировать все последующие (или предыдущие) элементы массива.

Как видно из рис. 9-4, для включения этих классов нижнего уровня в иерархию основных абстракций мы используем агрегацию. Более точно, диаграмма показывает, что мы используем физическое включение по значению с защищенным Доступом, которое означает, что это низкоуровневое представление доступно только подклассам и друзьям. На раннем этапе проектирования мы хотели воспользоваться примесями и сделать **unbounded** и **Bounded** защищенными суперклассами.

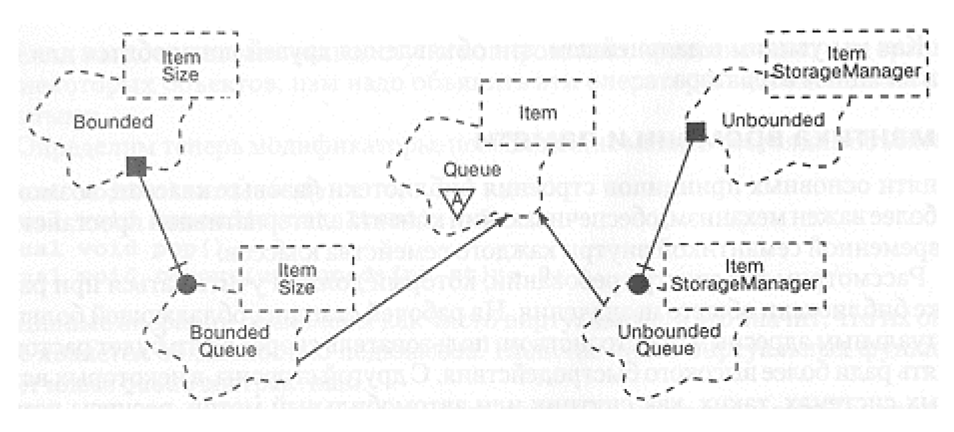


Рис. 9-4. Ограниченная и неограниченная формы

Мы в конце концов отказались от такого варианта, так как он достаточно труден для понимания, и к тому же нарушает лакмусов принцип наследования: **BoundedQueue**, по крайней мере, с точки зрения типа данных, не является частным случаем класса **Bounded**.

Отметим также, что работа с двумя формами требует присутствия второго аргумента в их шаблоне. Для ограниченной формы - это беззнаковое целое число **Size**, обозначающее статический размер объекта. Для неограниченной формы - это класс **StorageManager**, ответственный за политику размещения в памяти. Мы рассмотрим его работу в следующем разделе.

Протокол обоих классов поддержки должен быть, с одной стороны, достаточным для обеспечения работы конкретных подклассов, а с другой стороны, универсальным, чтобы гарантировать выполнение ответственности всех других структур в библиотеке. В целях компактности и быстродействия мы не включили в описание классов **Unbounded** и **Bounded** ни одной

виртуальной функции. По этой причине мы не можем объединить их одним суперклассом, несмотря на то, что они имеют общий протокол; кроме того, мы не можем надлежащим образом построить на их базе иерархию подклассов. В данном случае гибкость приносится в жертву производительности. По той же причине мы решаем сделать ряд функций встроенными;

хорошими кандидатами на это обычно являются селекторы, особенно те, которые возвращают простые переменные.

Рассмотрим, например, описание класса **Bounded**:

```
template<class Item, unsigned int Size>
class Bounded {
public:
	Bounded();
	Bounded(const Bounded<Item, Size>&);
	~Bounded();
	Bounded< Item, Size>& operator=(const Bounded<Item,
Size>&);
	int operator==(const Bounded<Item, Size>&) const;
	int operator!=(const Bounded<Item, Size>&) const;
	const Item& operator[](unsigned int index) const;
	Item& operator[](unsigned int index);
	void clear();
	void  insert(const      Item&);
	void  insert(const      Item&, unsigned int before);
	void  append(const      Item&);
	void  append(const      Item&, unsigned int after);
	void  remove(unsigned int at);
	void  replace(unsigned int at, const Item&);
	unsigned int available() const;
	unsigned int length() const;
	const Item& first() const;
	const Item& last() const;
	const Item& itemAt(unsigned int) const;
	Item& itemAt(unsigned int);
	int location(const Item&) const;
	static void* operator new(size_t);
	static void operator delete(void*, size_t);
protected:
	Item rep[Size] ;
	unsigned int start;
	unsigned int stop;
	unsigned int expandLeft(unsigned int from);
	unsigned int expandRight(unsigned int from);
	void shrinkLeft(unsigned int from);
	void shrinkRight(unsigned int from);
};
```

Объявление класса следует схеме, описанной ранее. Каким образом мы пришли именно к такому решению? Если честно, то на 80% это результат чистого проектирования классов, которое рассматривалось в главе 6. Затем интерфейс дорабатывался в соответствии с результатами пробного использования класса совместно с рядом основных абстракций системы. Основная трудность при эволюции состояла в идентификации подходящих примитивных операций, которые должны использоваться при работе с набором различных структур.

Сердцем класса является защищенный массив **rep** постоянного размера **Size**. Рассмотрим следующее объявление:

```
Bounded<char, 100U> charSequence;
```

При создании соответствующего объекта в стеке образуется массив постоянного размера из 100 элементов. Защищенные члены класса **start** и

stop (индексы в этом массиве) указывают начало и конец последовательности. Тем самым мы использовали кольцевой буфер данных. Добавление нового элемента в начало или в конец последовательности не потребует перемещения данных, а добавление элемента в середину массива приводит к копированию не более чем половины его элементов.

Проектирование ограниченного и неограниченного классов поддержки затрагивает также некоторые тонкие вопросы, касающиеся использования ссылок (мы упоминали о них в главе 3). Нам придется еще раз коснуться этой темы, и не только потому, что она имеет прямое отношение к разработке интерфейса параметризованных классов, но и потому, что данные вопросы сами по себе представляют значительный интерес для проектировщика любой более или менее нетривиальной библиотеки.

В C++ ссылки являются механизмом, позволяющим улучшить производительность. Однако пользоваться ими следует предельно осторожно во избежание нарушения корректного доступа к оперативной памяти. В данной библиотеке мы используем ссылки для ускорения работы при передаче аргументов функциям-членам. Это касается, например, класса **Bounded**, где подобным образом передаются ссылки на объекты классов **Bounded** и **Item**. Ссылки, как правило, не используются для передачи примитивных объектов (например, целых чисел в описании функции-члена **itemAt**) - программа от этого будет работать только медленнее. Кроме того, семантика языка C++ порождает некоторые опасности при манипулировании с временными объектами.

Все наши структуры, однако, содержат в качестве элементов не ссылки, а значения, что исключает возникновение ссылок на временные объекты в стеке при работе программы. По той же причине мы отказались от хранения указателей на элементы структур, так как это вызывает крайне нестабильное поведение системы при инстанцировании шаблона встроенными типами данных. Подобные вопросы чрезвычайно существенны при проектировании сред разработки, включающих в себя параметризованные классы, так как пользователь может инстанцировать шаблон произвольным типом данных. При использовании ссылок существуют, вообще говоря, три случая, и нам придется при создании библиотеки постараться найти определенный баланс между ними.

Во-первых, встроенные типы данных можно без труда передавать по ссылке и копировать. Объявив типы аргумента постоянными ссылками, можно избежать неприятностей, связанных с появлением временных структур, возникающих при приведении типов [12].

Во-вторых, типы данных, определенные пользователем, также можно передавать по ссылке и копировать, но только в том случае, когда для них определены копирующий конструктор и оператор присваивания. Ссылки можно использовать в полиморфных операциях (передавая объект производного класса вместо объявленного при инстанцировании), но копирование не будет полиморфным. В результате объект будет "срезан" до размеров своего базового класса.

В-третьих, при полиморфном использовании библиотеки встретится инстан-цирование шаблонов указателями на базовый класс. Хотя передача указателей по ссылке может и не улучшить производительность, но копирование указателей в представление сохраняет полиморфизм производных объектов.

Например, для класса **BoundedQueue** мы можем написать следующее:

```
class Event ... typedef Event* EventPtr;
BoundedQueue<int, 10U> intQueue;
BoundedQueue<Event, 50U> eventQueuel;
BoundedQueue<EventPtr, 100U> eventQueue2;
```

С помощью объекта класса **eventQueuel** можно спокойно создавать очереди событий, однако при добавлении в очередь экземпляра любого

подкласса **Event** произойдет "срезка", и полиморфное поведение такого экземпляра будет потеряно. С другой стороны, объект класса **eventQueue**2 содержит указатели на объекты класса **Event**, поэтому проблема "срезки" не возникает.

Наше решение, касающееся хранения внутри структур значений, а не ссылок, предъявляет определенные требования к конструкторам и деструкторам элементов. В частности, классы, используемые для инстанцирования структуры, должны, по крайней мере, иметь конструктор по умолчанию, копирующий конструктор и оператор присваивания. Кроме того, в некоторых случаях элементы не могут быть уничтожены сразу после удаления из структуры. В ограниченной форме, например, элементы (хранящиеся в массивах) не уничтожаются до уничтожения всей структуры.

Посмотрим, как можно использовать класс **Bounded** при формировании конкретного класса **BoundedQueue**. Отметим, что абстракция **BoundedQueue** содержит защищенный элемент **rep** класса **Bounded**.

```
template<class Item, unsigned int Size>
class BoundedQueue : public Queue<Item> {
public:
      BoundedQueue();
      BoundedQueue(const BoundedQueue< Item, Size>&);
      virtual ~BoundedQueue() ;
      virtual Queue<Item>& operator=(const Queue<Item>&);
      virtual Queue<Item>& operator=(const BoundedQueue<Item,
      Size>&) ;
      virtual int operator==(const Queue<Item>&) const;
      virtual int operator" (const BoundedQueue<Item, Size>&)
const;
      int operator!=(const BoundedQueue< Item, Size>&) const;
      virtual void clear();
      virtual void append(const Item&);
      virtual void pop();
      virtual void remove(unsigned int at);
      virtual unsigned int available() const;
      virtual unsigned int length() const;
      virtual int isEmpty() const;
      virtual const Item& front() const;
      virtual int location(const Item&) const;
protected:
      Bounded< Item, Size> rep;
      virtual void purge();
      virtual void add(const Item&);
      virtual unsigned int cardinality() const;
      virtual const Item& itemAt(unsigned int) const;
      static void* operator new(size_t);
      static void operator delete(void*, size_t);
};
```

Основная задача данного класса - завершить протокол, определенный в базовом классе. Часто это означает немного больше, чем простая передача обязанности классу более низкого уровня **Bounded**, как предлагается в следующей реализации:

```
template<class Item, unsigned int Size>
unsigned int BoundedQueue<Item, Size>::length() const
{
      return rep.length() ;
}
```

Отметим, что в описание класса **BoundedQueue** включены некоторые дополнительные операции, которых нет в его суперклассе. Добавлен селектор

available, возвращающий количество свободных элементов в структуре (вычисляется как разность **Size - length**()). Эта операция не включена в описание базового класса главным образом из-за того, что для неограниченной модели вычисление свободного места не очень осмысленно. Мы также переопределили оператор присваивания и проверку равенства. Как уже отмечалось ранее, это позволяет применить более эффективные алгоритмы по сравнению с базовым классом, так как подклассы лучше знают, что и как делать. Добавленные операторы **new** и **delete** определены в защищенной части класса, чтобы лишить клиентов возможности произвольно динамически размещать экземпляры **BoundedQueue** (что согласуется со статической семантикой этой конкретной формы).

Класс **Unbounded** имеет, в существенном, тот же протокол, что и класс **Bounded**, однако его реализация совершенно другая.

```
template<class Item, class StorageManager>
class Unbounded {
public:
…
protected:
	Node<Item, StorageManager>* rep;
	Node<Item, StorageManager>* last;
	unsigned int size;
	Node<Item, StorageManager>* cache;
	unsigned int cacheIndex;
};
```

Форма **Unbounded** реализует очередь как связный список узлов, где узел (**Node**) реализован следующим образом:

```
template<class Item, class StorageManager>
class Node {
public:
	Node(const Item& i,
	Node<ltem, StorageManager>* previous, Node<Item,
StorageManager>* next);
	Item item;
	Node<Item, StorageManager>* previous;
	Node<ltem, StorageManager>* next;
	static void* operator new(size_t);
	static void operator delete(void*, size_t);
};
```

Основная задача этого класса - управлять одним элементом списка и указателями на предыдущий и следующий узлы. Данная абстракция отнесена к категории классов поддержки, к ней не имеют доступ внешние пользователи, и поэтому мы решили несколько ослабить наши традиционные строгие требования к инкапсуляции, сделав все элементы класса открытыми и жертвуя таким образом безопасностью ради эффективности.

Помня, что классы **Bounded** и **Unbounded** имеют практически идентичный внешний протокол, а, значит, их функциональные свойства во многом подобны, можно предположить, что и реализация будет схожей. Однако различие во внутреннем представлении классов приводит к существенно различной пространственно-временной семантике. Манипуляции с узлами связанного списка, например, осуществляются очень быстро, однако процедура нахождения нужного элемента будет занимать время порядка *O(n)*. Поэтому наше представление кэширует последний узел, к которому было обращение, в надежде, что следующее обращение будет либо к этому же узлу, либо к его соседям. Схема же, базирующаяся на массивах, дает низкое быстродействие (в худшем случае порядка *O(n/2)* если элемент расположен в середине массива) при добавлении или удалении элементов, однако обеспечивает высокую скорость поиска (порядка *O(1)*).

Управление памятью

Задача управления памятью возникает для неограниченных форм реализации. В этом случае разработчик библиотеки должен определить политику выделения и освобождения памяти из кучи при осуществлении операций над узлами. Наивный подход просто использует глобальные функции **`new`** и **`delete`**, что не может обеспечить достаточной производительности системы. Кроме того, на некоторых компьютерных платформах управление памятью крайне усложнено (например, при наличии сегментированного адресного пространства в некоторых операционных системах персональных компьютеров) и требует разработки специальной стратегии, жестко привязанной к определенной операционной среде. Для нашей библиотеки надо четко выделить подсистему управления памятью.

На рис. 9-5 приведен выбранный для данной библиотеки механизм управления памятью.[24] Рассмотрим сценарий, иллюстрацией которого служит данная диаграмма:

• Клиент (**`aClient`**) вызывает операцию добавления (**`append`**) для экземпляра класса **`UnboundedQueue`** (более точно, экземпляра класса, инстанцированного из **`UnboundedQueue`**).

• **`UnboundedQueue`**, в свою очередь, передает выполнение операции своему элементу **`rep`**, который является экземпляром класса **`unbounded`**.

• **`Unbounded`**, вызывая свою статическую функцию **`new`**, выделяет необходимый объем адресного пространства для размещения нового экземпляра **`Node`**.

• Этот экземпляр **`Node`**, в свою очередь, делегирует ответственность за выделение памяти своему **`StorageManager`**, который доступен классу, инстанци-руемому из **`UnboundedQueue`** (и, следовательно, классам **`Unbounded`** и **`Node`**), как аргумент шаблона. **`StorageManager`** разделяется всеми экземплярами и служит для обеспечения последовательной политики выделения памяти на уровне класса.

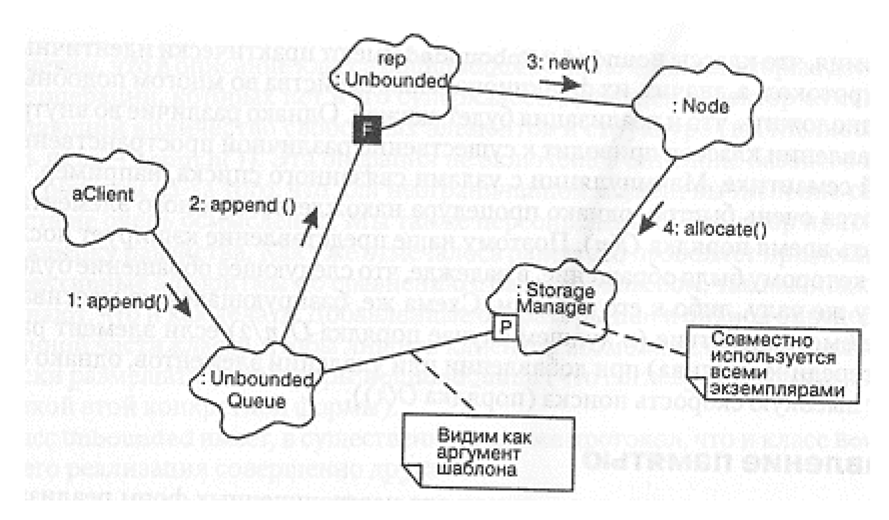


Рис. 9-5. Механизм управления памятью

Передавая **`StorageManager`** в качестве аргумента всем неограниченным структурам, мы четко отделяем политику организации доступа к памяти от ее реализации и даем пользователям возможность добавлять в программу свои собственные концепции управления памятью, не меняя при этом содержания библиотеки. Это классический пример того, как можно добиться открытости программной системы через инстанцирование, не прибегая к наследованию.

Единственное требование, предъявляемое к вариантам **`StorageManager`**, заключается в необходимости сохранения единого

протокола. В частности, все они должны содержать открытые функции-члены **allocate** и **deallocate**, предназначенные соответственно для выделения и освобождения памяти. Рассмотрим в качестве примера простейший вариант такого класса:

```
class Unmanaged {
public:
	static void* allocate(size_t s)
	{return ::operator new(s);} static void deallocate(void*
p, size_t) {::operator delete(p);}
private:
	Unmanaged() {}
	Unmanaged(Unmanaged&) {}
	void operator=(Unmanaged&) {}
	void operator==(Unmanaged&) {}
	void operator!=(Unmanaged&) {}
};
```

Обратите внимание на идиому, которая применяется, чтобы пользователь не мог копировать, присваивать и сравнивать экземпляры данного класса.

Протокол класса **Unmanaged** реализован через встроенные вызовы глобальных операторов **new** и **delete**. Мы назвали данную абстракцию **Unmanaged**, не требующей управления, так как она фактически не представляет собой ничего нового, а просто повторяет уже существующий системный механизм. Требующей управления названа другая абстракция, реализующая гораздо более эффективный алгоритм. В соответствии с этим алгоритмом память под узлы выделяется из некоего общего пула памяти. Если узел не используется, он помечается как свободный. Если возникает необходимость в новом узле, используется один из списка свободных. Выделение новой памяти из кучи происходит только в случае, если этот список пуст. Таким образом, часто удается избежать обращения к сервисным функциям операционной системы: выделение памяти сводится лишь к манипулированию указателями, что гораздо быстрее.[25]

При желании можно еще улучшить наш механизм, например, введя новую операцию для выделения памяти заранее, до того, как она понадобится. И наоборот, в определенных ситуациях, когда неиспользованных участков становится слишком много, можно дефрагментировать пул, и вернуть освободившуюся память в кучу. Можно предусмотреть операцию, позволяющую пользователю определить размер кластера памяти, и, таким образом, настроить класс под конкретное приложение.

В соответствии с приведенными выше соображениями, соответствующий класс поддержки можно определить следующим образом:

```
class Pool {
public:
	Pool(size_t chunkSize);
	~Pool();
	void* allocate(size_t);
	void deallocate(void*, size_t);
	void preallocate(unsigned int numberOfChunks);
	void reclaimUnusedChunks();
	void purgeUnusedChunks();
	size_t chunkSize() const;
	unsigned int totalChunks() const;
	unsigned int numberOfDirtyChunks() const;
	unsigned int numberOfUnusedChunks() const;
protected:
	struct Element ... struct Chunk ...
	Chunk* head;
	Chunk* unusedChunks;
```

```
        size_t repChunkSize;
        size_t usableChunkSize;
        Chunk* getChunk(size_t s);
};
```

Описание содержит два вложенных класса **Element** и **chunk** (отрезок). Каждый экземпляр класса **Pool** управляет связным списком объектов **chunk**, представляющих собой отрезки "сырой" памяти, но трактуемых как связные списки экземпляров класса **Element** (это один из важных аспектов, управляемых классом **pool**). Каждый отрезок может отводиться элементам разного размера и для эффективности мы сортируем список отрезков в порядке возрастания их размеров. Менеджер памяти может быть определен следующим образом:

```
class Managed {
public:
        static Pool& pool;
        static void* allocate(size_t s)
        {return pool.allocate(s); } static void deallocate(void*
p, size_t s)
        {pool.deallocate(p, s);}
private:
        Managed() {}
        Managed(Managed&) {}
        void operator=(Managed&) {}
        void operator==(Managed&) {}
        void operator!=(Managed&) {} };
```

Этот класс имеет тот же внешний протокол, что и **Unmanaged**. Из-за того, что в C++ шаблоны сознательно недостаточно четко определены, соответствие данному протоколу проверяется только при трансляции инстанцированного класса типа **UnboundedQueue**, в тот момент, когда конкретный класс сопоставляется с формальным аргументом **StorageManager**.

Объект класса **Pool**, принадлежащий классу **Managed**, является статическим. Это позволяет нескольким конкретным структурам (требующим управления) делить между собой единый пул памяти. Различные структуры, не требующие управления, могут, конечно, определить своего менеджера и свой пул памяти, предоставляя таким образом разработчику полный контроль над политикой выделения памяти.

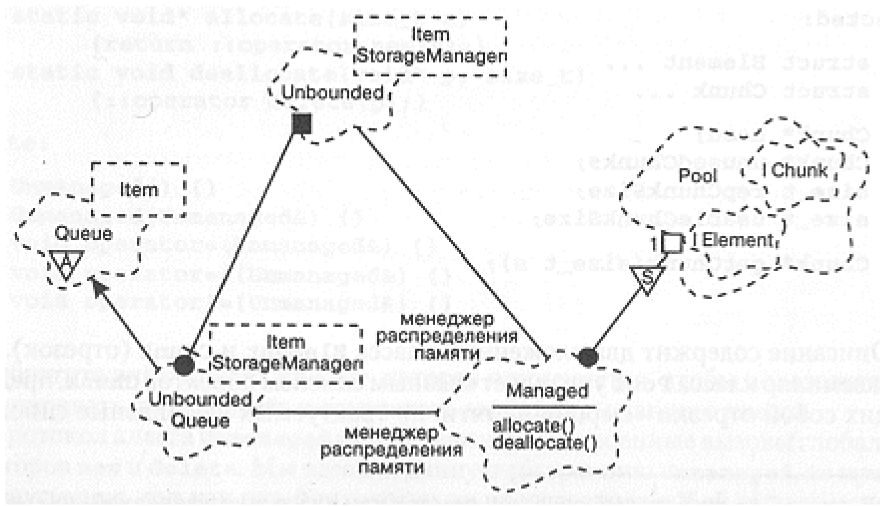


Рис. 9-6. Классы управления памятью

На рис. 9-6 приведена диаграмма классов, иллюстрирующая схему взаимодействия различных классов, обеспечивающих управление памятью.

Мы показали только ассоциативную связь между классом **Managed** и его клиентами **Unbounded** и **UnboundedQueue**; эта ассоциация будет уточнена при конкретном инстанцирова-нии классов.

Физическая компоновка классов поддержки тоже является частью архитектурного решения. Рис. 9-7 иллюстрирует их модульную архитектуру. Мы выбрали именно такую схему, чтобы изолировать классы, которые, по-видимому, будут чаще всего подвергаться изменениям.

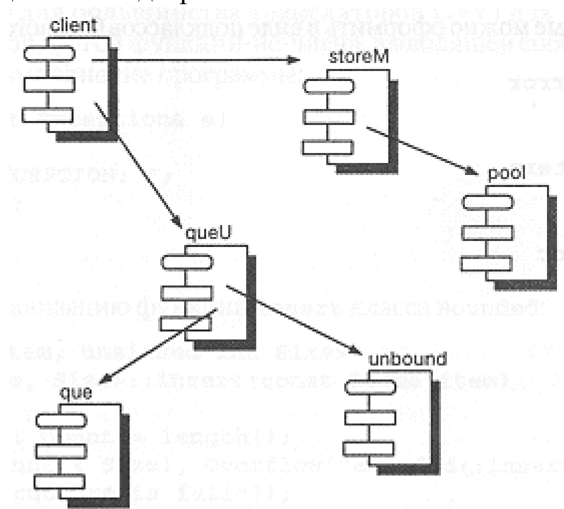


Рис. 9-7. Модули управления памятью

Исключения

Несмотря на то, что язык C++ можно заставить соблюдать многие статические предположения (нарушение которых повлечет ошибку компиляции), для выявления динамических нарушений (таких, как попытка добавить элемент к полностью заполненной ограниченной очереди или удалить элемент из пустого списка) приходится использовать и другие механизмы. В данной библиотеке используются средства обработки исключений, предоставляемые C++ [14]. Наша архитектура включает в себя иерархию классов исключений и, отдельно от нее, ряд механизмов по выявлению таких ситуаций.

Начнем с базового класса **Exception** (исключение), обладающего несложным протоколом:

```
class Exception {
public:
        Exception(const char* name, const char* who, const char*
what);
        void display() const;
        const char* name() const;
        const char* who() const;
        const char* what() const;
protected:
…
};
```

Каждой особой ситуации можно сопоставить имя ее источника и причину возникновения. Кроме того, мы можем обеспечить скрытые от клиентов средства для вывода информации об ошибке в соответствующий поток.

Анализ различных классов нашей библиотеки подсказывает возможные типы исключений, которые можно оформить в виде подклассов базового класса **Exception**:

- **ContainerError**
- **Duplicate**
- **IllegalPattern**
- **IsNull**
- **LexicalError**
- **MathError**
- **NotFound**
- **NotNull**
- **NotRoot**
- **Overflow**
- **RangeError**
- **StorageError**
- **Underflow**

Объявление класса **overflow** (переполнение) может выглядеть следующим образом:

```
class Overflow : public Exception {
public:
	Overflow(const char* who, const char* what)
	: Exception("Overflow", who, what) {}
};
```

Обязанность этого класса состоит лишь в знании своего имени, которое он передает конструктору суперкласса.

В данном механизме функции-члены классов библиотеки только возбуждают исключения; они не в состоянии перехватить исключение, главным образом, потому, что ни одна из них не может осмысленно отреагировать на эту ситуацию. По соглашению мы возбуждаем исключение при нарушении условий, предполагавшихся относительно некоторого состояния. Условие представляет собой обычное булевское выражение, которое должно быть истинным в нормальной ситуации. Чтобы упростить библиотеку, мы ввели следующую функцию, не принадлежащую ни одному из классов:

```
inline void _assert(int expression,const Exception&
exception)
{
	if (!expression)
	throw(exception);
}
```

Для эффективности мы определили эту функцию как встроенную. Преимущество подобной схемы состоит в том, что она локализует все исключения (в C++ throw имеет синтаксис вызова функции). Так, для трансляторов, которые до сих пор не поддерживают исключений, можно использовать специальную директиву (-D для большинства трансляторов C++) для переопределения вызова throw в вызов другой функции-не-члена, выводящей сообщение на экран и останавливающей выполнение программы:

```
void _catch(const Exception& e)
{
	cerr << "EXCEPTION: ";
	e.display();
	exit(l);
}
```

Рассмотрим реализацию функции **insert** класса **Bounded**:

```
template<class Item, unsigned int Size>
void Bounded<ltem, Size>::insert(const Item& item)
{
	unsigned int count = length();
	_assert((count < Size), Overflow("Bounded::Insert",
"structure is full"));
```

```
        if (!count)
        start = stop = 1;
        else
        {
              start--;
              if (!start)
              start = Size;
        }
        rep[start - 1] = item;
}
```

Предусмотрено, что в процессе выполнения функции проверяется, что размер структуры не превосходит максимально допустимого. Если это не так, возбуждается исключение **`Overflow`**.

Важнейшим преимуществом этого подхода является гарантия того, что состояние объекта, возбудившего исключение, не будет нарушено (не считая случая исчерпания оперативной памяти, когда уже в принципе ничего нельзя поделать). Любая функция, прежде чем произвести действия, способные изменить состояние объекта, проверяет предположение. В приведенной выше функции **`insert`**, например, прежде, чем добавить элемент в массив, мы сначала вызываем селектор (который не может вызвать изменения состояния объекта), затем проверяем все предусловия функции и лишь затем изменяем состояние объекта. Мы скрупулезно придерживались подобного стиля при реализации всех функций и настоятельно советуем не отходить от него при конструировании подклассов, основанных на нашей библиотеке.

Рис. 9-8 иллюстрирует схему взаимодействия классов, обеспечивающих реализацию механизма обработки исключений.

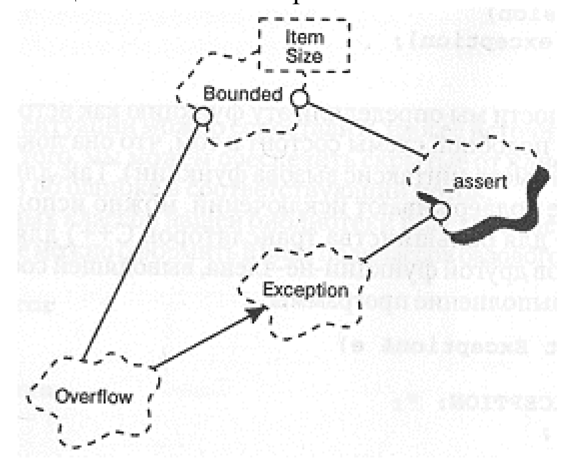


Рис. 9-8. Классы обработки исключений

Итерация

Итерация - это еще один архитектурный шаблон нашей библиотеки. В главе 3 уже отмечалось, что итератор представляет собой операцию, обеспечивающую последовательный доступ ко всем частям объекта. Оказывается, такой механизм нужен не только пользователям, он необходим и при реализации самой библиотеки, в частности, ее базовых классов.

При этом перед нами стоял выбор: можно было определять итерации как часть протокола объектов или создавать отдельные объекты, ответственные за итеративный опрос других структур. Мы выбрали второй подход по двум причинам:

- Наличие выделенного итератора классов позволяет одновременно проводить несколько просмотров одного и того же объекта.
- Наличие итерационного механизма в самом классе несколько нарушает его инкапсуляцию; выделение итератора в качестве отдельного

механизма поведения способствует достижению большей ясности в описании класса.

Для каждой структуры определены две формы итераций. Активный итератор требует каждый раз от клиента явного обращения к себе для перехода к следующему элементу. Пассивный итератор применяет функцию, предоставляемую клиентом, и, таким образом, требует меньшего участия клиента.1 Чтобы обеспечить безопасность типов, для каждой структуры создаются свои итераторы.

Рассмотрим в качестве примера активный итератор для класса **Queue**:

```
template <class Item> class QueueActiveIterator {
public:
QueueActiveIterator(const Queue<Item>&);
~QueueActiveIterator();
```

Пассивный итератор реализует "применяемую" функцию. Эта идиома обычно используется в функциональных языках программирования.

```
void reset();
int next() ;
int isDone() const;
const Item* currentltem() const;
protected:
const Queue<Item>& queue;
int index;
};
```

Каждому итератору в момент создания ставится в соответствие определенный объект. Итерация начинается с "верха" структуры, что бы это ни значило для данной абстракции.

С помощью функции **currentItem** клиент может получить доступ к текущему элементу; значение возвращаемого указателя может быть нулевым в случае, если итерация завершена или если массив пуст. Переход к следующему элементу последовательности происходит после вызова функции **next** (которая возвращает О, если дальнейшее движение невозможно, как правило, из-за того, что итерация завершена). Селектор **isDone** служит для получения информации о состоянии процесса: он возвращает 0, если итерация завершена или структура пуста. Функция **reset** позволяет осуществлять неограниченное количество итерационных проходов по объекту.

Например, при наличии следующего объявления:

```
BoundedQueue<NetworkEvent> eventQueue;
```

фрагмент кода, использующий активный итератор для захода в каждый элемент очереди, будет выглядеть так:

```
QueueActiveIterator<NetworkEvent> iter(eventQueue);
while (!iter.isDone()) {
iter.currentItem()->dispatch();
iter.next();
}
```

Итерационная схема, приведенная на рис. 9-9, иллюстрирует данный сценарий работы и, кроме того, раскрывает некоторые детали реализации итератора. Рассмотрим их более подробно.

Конструктор класса **QueueActiveIterator** сначала устанавливает связь между итератором и конкретной очередью. Затем он вызывает защищенную функцию **cardinality**, которая определяет количество элементов в очереди. Таким образом, конструктор можно описать следующим образом:

```
template<class Item>
QueueActiveIterator<ltem>::QueueActiveIterator(const
Queue<Item>& q)
:queue(q), index(q.cardinality() ? O : -1) {}
```

Класс **QueueActiveIterator** имеет доступ к защищенной функции **cardinality** класса **Queue**, поскольку числится в дружественных ему.

Операция итератора **isDone** проверяет принадлежность текущего индекса допустимому диапазону, который определяется количеством элементов очереди:

```
template<class Item>
```

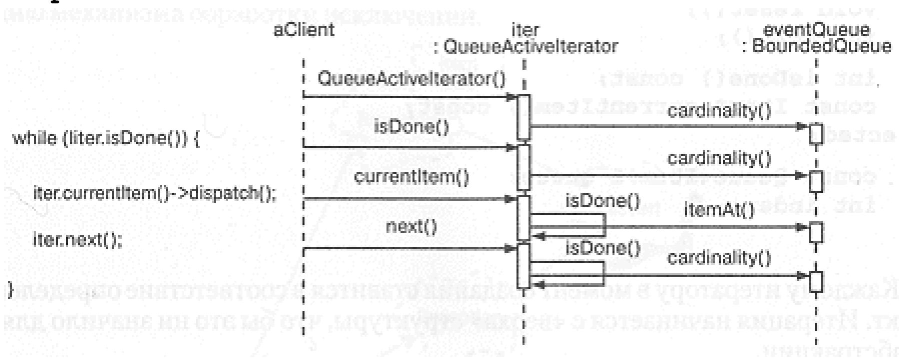


Рис. 9-9. Механизм итерации

```
int   QueueActiveIterator<Item>::isDone() const
{
      return ((index < 0) II (index >= queue.cardinality()));
}
```

Функция **currentItem** возвращает указатель на элемент, на котором остановился итератор. Реализация итератора в виде индекса объекта в очереди дает возможность в процессе итераций без труда добавлять и удалять элементы из очереди:

```
template<class Item>
const Item* QueueActiveIterator<Item>::currentItem () const
{
      return isDone() ? 0 : &queue.itemAt(index);
}
```

При выполнении данной операции итератор снова вызывает защищенную функцию очереди, на сей раз **itemAt**. Кстати, **currentItem** можно использовать для работы как с ограниченной, так и с неограниченной очередью. Для ограниченной очереди **itemAt** просто возвращает элемент массива по соответствующему индексу. Для неограниченной очереди операция **itemAt** будет осуществлять проход по связному списку. Правда, как мы помним, класс **Unbounded** хранит информацию о последнем элементе, к которому было обращение, поэтому переход к следующему за ним элементу очереди (что и происходит при продвижении итератора) будет достаточно простым.

Операция **next** увеличивает значение текущего индекса на единицу, что соответствует переходу к следующему элементу очереди, а затем проверяет допустимость нового значения индекса:

```
template<class Item>
int QueueActiveIterator<Item>::next()
{
      index++;
      return !isDone();
}
```

Итератор, таким образом, в процессе своей работы вызывает две защищенные функции класса **Queue**: **cardinality** и **itemAt**. Определив эти функции как чисто виртуальные, мы передали ответственность за их конкретную оптимальную реализацию классам, производным от **Queue**.

Ранее отмечалось, что одна из основных задач наших архитектурных решений заключается в том, чтобы дать возможность клиенту копировать, присваивать и проверять на равенство экземпляры абстрактного базового

класса, даже если они имеют различное представление. Эта возможность достигается за счет использования итераторов и некоторых служебных функций, позволяющих просматривать структуры независимо от их представления. Например, оператор присваивания для класса **Queue** можно определить следующим образом:

```
template<class Item>
Queue<Item>& Queue<Item>::operator=(const Queue<Item>& q)
{
	if (this == &q)
	return *this;
	((Queue<Item>&)q).lock();
	purge();
	QueueActiveIterator<Itea> iter(q);
	while (!iter.isDone()) {
	add(*iter.currentltem());
	iter.next();
	} ((Queue<Item>&)q).unlock();
	return *this;
}
```

В данном алгоритме используется идиома блокирования, которая более подробно рассмотрена в следующем разделе.

Присваивание осуществляется в порядке просмотра активным итератором структуры, определяемой аргументом **q**. Сначала защищенная служебная функция **purge** очищает очередь, а затем к ней с помощью другой защищенной служебной функции **add** последовательно добавляются новые элементы. Тот факт, что процесс итерации осуществляется с помощью полиморфных функций, дает возможность копировать, присваивать и проверять на равенство объекты, имеющие одинаковую структуру, но с разными представлениями.

Пассивный итератор, который также называют аппликатором, характеризуется тем, что он применяет определенную функцию к каждому элементу структуры. Для класса **Queue** пассивный итератор можно определить следующим образом:

```
template <class Item>
class QueuePassiveIterator {
Public:
	QueuePassiveIterator(const Queue<Item>&);
	~QueuePassiveIterator();
	int apply(int (*)(const Item&));
Protected:
	const Queue<Item>& queue;
};
```

Пассивный итератор действует на все элементы структуры за (логически) одну операцию. Таким образом, функция **apply** последовательно производит одну и ту же операцию над каждым элементом структуры, пока передаваемая итератору функция не возвратит нулевое значение или пока не будет достигнут конец структуры (в первом случае функция **apply** сама возвратит нулевое значение в знак того, что итерация не была завершена).

Синхронизация

При разработке любого универсального инструментального средства должны учитываться проблемы, связанные с организацией параллельных процессов. В операционных системах типа UNIX, OS/2 и Windows NT приложения могут запускать несколько "легких" процессов[26]. В большинстве случаев классы просто не смогут работать в такой среде без специальной доработки: когда две задачи взаимодействуют с одним и тем же объектом, они должны делать это согласованно, чтобы не разрушить состояния объекта. Как

уже отмечалось, существуют два подхода к задаче управления процессами; они находят свое отражение в существовании защищенной и синхронизированной форм класса.

При разработке данной библиотеки было сделано следующее предположение:

разработчики, планирующие использовать параллельные процессы, должны импортировать либо разработать сами по крайней мере класс **Semaphore** (семафор) для синхронизации легких процессов. Разработчики, которые не хотят связываться с параллельными процессами, будут свободны от необходимости поддерживать защищенные или синхронизованные формы классов (таким образом, не потребуется никаких дополнительных издержек). Защищенные и синхронизированные формы изолированы в библиотеке и основываются на своей внутренней реализации параллелизма. Единственная зависимость от локальной реализации сосредоточена в классе **Semaphore**, который имеет следующий интерфейс:

```
class Semaphore {
public:
	Semaphore ();
	Semaphore (const Semaphore&);
	Semaphore (unsigned int count);
	~Semaphore ();
	void seize(); // захватить void release(); // освободить
	unsigned int nonePending() const;
protected:
};
```

Так же, как и при управлении памятью, мы разделяем политику синхронизации процессов и ее реализацию. По этой причине в аргументы шаблона для каждой защищенной формы включен класс **Guard** (страж), ответственный за связь с локальной реализацией класса **Semaphore** или его эквивалента. Аргументы шаблона для каждой из синхронизированных форм содержат класс **Monitor**, который близок по своим функциональным свойствам к классу **Semaphore**, но, как будет видно в дальнейшем, обеспечивает более высокий уровень параллелизма процессов.

Как показано на рис. 9-3, защищенный класс является прямым подклассом своего конкретного ограниченного либо неограниченного класса и содержит в себе объект класса **Guard**. Все защищенные классы имеют общедоступные функции-члены **seize** (захватить) и **release** (освободить), позволяющие получить эксклюзивный доступ к объекту. Рассмотрим в качестве примера класс **GuardedUnboundedQueue**, производный от **UnboundedQueue**:

```
template<class Item, class StorageManager, class Guard>
class GuardedUnboundedQueue : public UnboundedQueue<Item,
StorageManager> {
public:
	GuardedUnboundedQueue ();
	virtual ~GuardedUnboundedQueue ();
	virtual void seize();
	virtual void release();
protected:
	Guard guard;
};
```

В нашей библиотеке предусмотрен интерфейс одного из предопределенных классов защиты: класса **semaphore**. Пользователи могут дополнить реализацию данного класса в соответствии с локальным определением легкого процесса.

На рис. 9-10 приведена схема работы данного варианта синхронизации; клиенты, использующие защищенные объекты, должны

придерживаться простого алгоритма: сначала захватить объект для эксклюзивного доступа, провести над ним нужную работу, и после ее окончания снять защиту (в том числе в тех случаях,

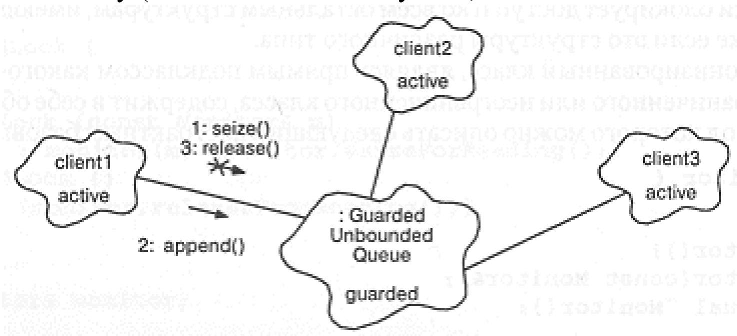


Рис. 9-10. Процессы защищенного механизма

когда возникла исключительная ситуация). Другая схема поведения рассматривается как социально неприемлемая, поскольку претензии одного агента не позволят правильно работать другим. Если мы, например, не снимем защиту после окончания работы с объектом, больше никто не сможет получить к нему доступ; попытка снятия защиты с объекта, к которому в данный момент никто не имел эксклюзивного доступа, также может привести к нежелательным последствиям. Игнорирование этого протокола просто безответственно, поскольку оно может разрушить состояние объекта, с которым одновременно работают несколько агентов.

Основное преимущество защищенной схемы - ее простота. В то же время для агентов, производящих операции над одним и тем же объектом, использование данной модели обуславливает необходимость выполнения определенных коллективных действий. Другая особенность защищенных форм состоит в том, что она дает возможность агентам выделять критически важные моменты, когда несколько операций, произведенных над объектом, будут гарантированно интерпретироваться как одна атомарная транзакция.

Подобно механизму управления памятью, сигнатура шаблона защищенной формы импортирует стража, а не превращает его в неизменяемую характеристику. Это позволяет пользователям ввести новую политику синхронизации. При использовании в качестве стража предопределенного класса **`Semaphore`**, стандартная политика синхронизации подразумевает, что каждому объекту ставится в соответствие свой семафор. Данное решение приемлемо только до тех пор, пока количество параллельных процессов не достигнет некоторого критического значения.

Альтернативный подход подразумевает возможность обслуживания одним семафором сразу нескольких защищенных объектов. Разработчику при этом нужно только создать новый класс-страж, имеющий тот же протокол, что и **`semaphore`** (но не обязательно являющийся его подклассом). Этот класс может содержать семафор в качестве статического члена; тогда семафор будет совместно использоваться всеми экземплярами класса. Инстанцируя защищенную форму с этим новым стражем, разработчик библиотеки вводит новую политику, поскольку все объекты инстанцированного класса пользуются общим стражем, вместо выделения отдельного стража каждому объекту. Преимущество данной схемы наиболее ясно проявляется, когда новый класс-страж используется для инстанцирования других структур:

все полученные объекты будут работать с одним и тем же стражем. Таким образом, на первый взгляд незначительное изменение политики приводит не только к уменьшению количества параллельных процессов, но также позволяет клиенту блокировать целую группу объектов, несвязанных напрямую. Захват одного объекта автоматически блокирует доступ и ко всем остальным структурам, имеющим того же стража, даже если это структуры различного типа.

Синхронизированный класс, являясь прямым подклассом какого-либо конкретного ограниченного или неограниченного класса, содержит в себе объект-монитор, протокол которого можно описать следующим абстрактным базовым классом:

```
class Monitor {
public:
	Monitor();
	Monitor(const Monitor&);
	virtual ~Monitor();
	virtual void seizeForReading() = 0;

	virtual void seizeForWriting() = 0;
	virtual void releaseFromBeadingt() = 0;
	virtual void releaseFromWritingt() = 0;
protected:
	…
};
```

С помощью мониторов можно реализовать два типа синхронизации:

• Одиночная Гарантирует семантику структуры в присутствии нескольких потоков управления, но с одним читающим или одним записывающим.

• Множественная Гарантирует семантику структуры в присутствии нескольких потоков управления, с несколькими читающими или одним записывающим.

Агент записи меняет состояние объекта; агенты записи вызывают функции-модификаторы. Агент чтения сохраняет состояние объекта; он вызывает только функции-селекторы. Как видно, множественная форма синхронизации обеспечивает наивысшую степень параллелизма процессов. Мы можем реализовать обе политики в виде подклассов абстрактного базового класса **Monitor**. Обе формы можно построить на основе класса **Semaphore**.

В отличие от защищенных форм, синхронизованные классы не содержат дополнительных функций-членов по сравнению со своим суперклассом: они просто переопределяют все виртуальные функции суперкласса. Семантика, вносимая синхронизированным классом, заставляет трактовать каждую такую функцию как атомарную транзакцию. В то время, как клиенты защищенного объекта должны для получения эксклюзивного доступа каждый раз явно захватывать и освобождать доступ, синхронизированные формы обеспечивают эксклюзивность доступа, не требуя специальных действий со стороны своих клиентов.

Это достигается с помощью механизма блокировки, схема работы которого приведена на рис. 9-11. Взаимодействие мониторов с экземплярами предопределенных классов **ReadLock** и **WriteLock** обеспечивает эксклюзивность вызова каждой функции-члена. В этом механизме блокировка использует либо семафор, либо монитор в качестве агента, ответственного за процесс синхронизации, а сама блокировка отвечает за захват этого агента при создании и освобождение при удалении. В качестве примера рассмотрим определение класса **ReadLock**:

```
class ReadLock {
public:
	ReadLock (const Monitor& m)
	: monitor(m) {monitor.seizeForReadingt);} ~ReadLock ()
	(monitor.releaseFromReading();}
private:
	Monitor& monitor;
};
```

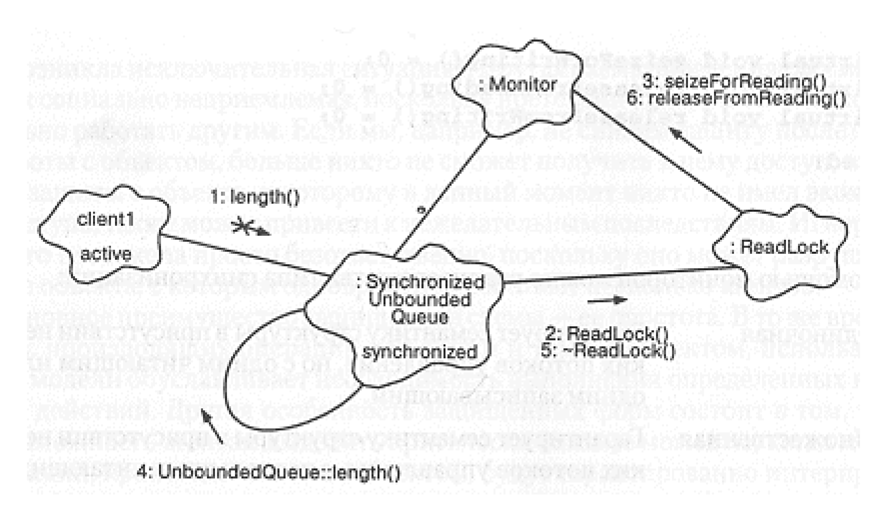


Рис. 9-11. Механизм блокировки

Определив блокировку и ее монитор как две отдельные абстракции, мы дали клиенту возможность использовать различные политики блокировки. Описание класса **WriteLock** аналогично, разница лишь в том, что он использует протокол монитора для записи.

Описания всех функций-членов синхронизированного класса используют блокировки для "оборачивания" операций, унаследованных из суперкласса. Рассмотрим в качестве примера реализацию функции **length** для синхронизированной неограниченной очереди:

```
template<class Item, class StorageManager, class Monitor>
unsigned int SynchronizedUnboundedQueue<Item, StorageManager,
        Monitor>::length() const
{
        ReadLock lock(monitor);
        return UnboundedQueue<Item, StorageManager>::length();
}
```

Данный фрагмент кода иллюстрирует механизм, приведенный на рис. 9-11. Как правило, объекты класса **ReadLock** используются для всех синхронизированных селекторов, а экземпляры **WriteLock** - для синхронизированных модификаторов. Простота и элегантность подобной архитектуры проявляется в том, что каждая функция представляет собой законченную операцию, в любом случае гарантирующую сохранность состояния объекта, причем без каких-либо явных действий со стороны агентов чтения/записи.

Действительно, клиенты, работающие с синхронизированными объектами, не должны придерживаться специальной последовательности действий, так как механизм синхронизации процессов поддерживается здесь в неявном виде. Это исключает появление ошибок типа неверной блокировки. Разработчику следует, однако, предпочитать защищенную форму синхронизированной, когда вызов нескольких функций нужно оформить как атомарную транзакцию; синхронизированная форма может гарантировать атомарность только отдельных функций-членов.

Наша архитектура обеспечивает синхронизированным формам отсутствие ситуаций типа "смертельное объятие". Например, операции присваивания объекта самому себе или сравнения его с самим собой потенциально опасны, так как требуют блокировки и левого и правого элементов выражения, которые в данном случае являются одним и тем же объектом. Будучи создан, объект не может изменить свою идентичность, поэтому тесты на самоидентичность выполнятся до блокировки какого-либо объекта. Именно поэтому описанный ранее оператор присваивания **operator=** включал такую проверку, как показывает следующая сокращенная запись:

```
template<class Item>
Queue<Item>& Queue<Item>::operator=(const Queue<Item>& q)
{
      if (this == &q)
      return *this;
}
```

Любые функции-члены, среди аргументов которых есть экземпляры класса, к которому они принадлежат, должны проектироваться так, чтобы обеспечивалась корректная схема блокировки этих аргументов. Наше решение базируется на полиморфизме двух служебных функций, **lock** и **unlock**, определенных в каждом абстрактном базовом классе. Каждый абстрактный базовый класс по умолчанию содержит заглушку для этих двух функций; синхронизированные формы обеспечивают захват и освобождение аргумента. Вот почему описанный ранее оператор присваивания **operator=** включал вызовы этих двух функций, как показывает следующая сокращенная запись:

```
      template<class Item>
      Queue<Item>& Queue<Item>::operator=(const Queue<Item>&
q)
      {
      ((Queue<Item>&)q).lock();
      ((Queue<Item>&)q),unlock();
      return *this;
      }
```

Явное приведение типа используется в данном случае для того, чтобы освободиться от ограничения **const** на аргумент.

9.3. Эволюция

Проектирование интерфейса классов

После того, как выработаны основные принципы построения архитектуры системы, остающаяся работа проста, но зачастую довольно скучна и утомительна. Следующий этап будет состоять в реализации трех или четырех семейств классов (таких, как очередь, множество и дерево) в соответствии с выбранной архитектурой, и в последующем их тестировании в нескольких приложениях[27].

Наиболее тяжелой частью данного этапа является создание подходящего интерфейса для каждого базового класса. И здесь, в процессе изолированной разработки отдельных классов (см. главу 6), нельзя забывать о задаче обеспечения глобального соответствия всех частей системы друг другу. В частности, для класса **Set** можно определить следующий протокол:

- **setHashFunction** Устанавливает функцию хеширования для элементов множества.
- **clear** Очищает множество.
- **add** Добавляет элемент к множеству.
- **remove** Удаляет элемент из множества.
- **setUnion** Объединяет с другим множеством.
- **intersection** Находит пересечение с другим множеством.
- **difference** Удаляет элементы, которые содержатся в другом множестве.
- **extent** Возвращает количество элементов в множестве.
- **isEmpty** Возвращает 1, если множество пусто.
- **isMember** Возвращает 1, если данный элемент принадлежит множеству.

• **isSubset** Возвращает 1, если множество является подмножеством другого множества.

• **isProperSubset** Возвращает 1, если множество является собственным подмножеством другого множества.

Подобным же образом можно определить протокол класса BinaryTree:

• **clear** Уничтожает дерево и всех его потомков.

• **insert** Добавляет новый узел в корень дерева.

• **append** Добавляет к дереву потомка.

• **remove** Удаляет потомка из дерева.

• **share** Структурно делит данное дерево.

• **swapChild** Переставляет потомка с деревом.

• **child** Возвращает данного потомка.

• **leftChild** Возвращает левого потомка.

• **rightChild** Возвращает правого потомка.

• **parent** Возвращает родителя дерева.

• **setItem** Устанавливает элемент, ассоциированный с деревом.

• **hasChildren** Возвращает 1, если у дерева есть потомки.

• **isNull** Возвращает 1, если дерево нулевое.

• **isShared** Возвращает 1, если дерево структурно разделено.

• **isRoot** Возвращает 1, если дерево имеет корень.

• **itemAt** Возвращает элемент, ассоциированный с деревом.

Для схожих операций мы используем схожие имена. При разработке интерфейса мы также проверяем полученное решение на соответствие критериям достаточности, полноты и примитивности (см. главу 3).

Классы поддержки

При реализации класса, ответственного за манипуляции с текстовыми строками, мы столкнулись с тем, что возможностей, предоставляемых классами поддержки **Bounded** и **Unbounded**, явно недостаточно. Ограниченная форма, в частности, оказывается неэффективной для работы со строками с точки зрения памяти, так как мы должны инстанцировать эту форму в расчете на максимально возможную строку, и следовательно понапрасну расходовать память на более коротких строках. Неограниченная форма, в свою очередь, неэффективна с точки зрения быстродействия: поиск элемента в строке может потребовать последовательного перебора всех элементов связного списка. По этим причинам нам пришлось разработать третий, "динамический" вариант:

• Динамический Структура хранится в "куче" в виде массива, длина кото-рого может уменьшаться или увеличиваться.

Соответствующий класс поддержки **Dynamic** представляет собой промежуточный вариант по отношению к ограниченному и неограниченному классам, обеспечивающий быстродействие ограниченной формы (возможно прямое индексирование элементов) и эффективность хранения данных, присущую неограниченной форме (память выделяется только под реально существующие элементы).

Ввиду того, что протокол данного класса идентичен протоколу классов **Bounded** и **Unbounded**, добавление к библиотеке нового механизма не составит большого труда. Мы должны создать по три новых класса для каждого семейства (например, **DynamicString**, **GuardedDynamicString**, И **SynchronizedDynamicString**). Таким образом, мы вводим следующий класс поддержки:

```
template<class Item, class StorageManager> class Dynamic
{ Public:
Dynamic(unsigned int chunkSize);
Protected:
Item* rep;
unsigned int size;

unsigned    int totalChunks;
unsigned    int chunkSize;
unsigned    int start;
unsigned    int stop;
void resize(unsigned int currentLength,
     unsigned int newLength, int preserve     - 1);
unsigned    int expandLeft(unsigned int from);
unsigned    int expandRight(unsigned int from);
void shrinkLeft(unsigned int from);
void shrinkRight(unsigned int from);
};
```

Последовательности разбиваются на блоки в соответствии с аргументом конструктора **chunkSize**. Таким образом, клиент может регулировать размер будущего объекта.

Из интерфейса видно, что класс **Dynamic** имеет много общего с классами **Bounded** и **Unbounded**. Отличия в реализации трех типов классов каждого семейства будут минимальны.

Займемся теперь классом ассоциативных массивов. Его реализация потребует новой переработки ограниченной, динамической и неограниченной форм. В частности, поиск элемента в ассоциативном массиве требует слишком много времени, если его приходится вести перебором всех элементов. Но производительность можно значительно увеличить, используя открытые хеш-таблицы.

Абстракция открытой хеш-таблицы проста. Таблица представляет собой массив последовательностей, которые называются клетками. Помещая в таблицу новый элемент, мы сначала генерируем хеш-код по этому элементу, а затем используем код для выбора клетки, куда будет помещен элемент. Таким образом, открытая хеш-таблица делит длинную последовательность на несколько более коротких, что значительно ускоряет поиск.

Соответствующую абстракцию можно определить следующим образом:

```
template<class Item, class Value, unsigned int Buckets,
class Container> class Table { public:
Table(unsigned int (*hash)(const Item&))
void setHashFunction(unsigned int (*hash)(const Item&));
void clear();
int bind(const Item&, const Value&);
int rebind(const Item&, const Value&);
int unbind(const Item&);
Container* bucket(unsigned int bucket);
unsigned int extent() const;
int isBound(const Item&) const;
const Value* valueOf(const Item&) const;
const Container *const bucket(unsigned int bucket)
const;
protected:
Container rep[Buckets];

};
```

Использование класса **Container** в качестве аргумента шаблона позволяет применить абстракцию хеш-таблицы вне зависимости от типа

конкретной последовательности. Рассмотрим в качестве примера (сильно упрощенное) объявление неограниченного ассоциативного массива, построенного на базе классов **Table** и **Unbounded**:

```
template<class Item, class Value, unsigned int Buckets,
class StorageManager> class UnboundedMap : public
Map<Item, Value> { public:
UnboundedMap();
virtual int bind(const Item&, const Value&);
virtual int rebind(const Item&, const Value&);
virtual int unbind(const Item&);
protected:
Table<Item, Value, Buckets, Unbounded<Pair<Item, Value>,
StorageManager> > rep;
};
```

В данном случае мы истанцируем класс **Table** контейнером **unbounded**. Рис. 9-12 иллюстрирует схему взаимодействия этих классов.

В качестве свидетельства общей применимости этой абстракции мы можем использовать класс **Table** при реализации классов множеств и наборов.

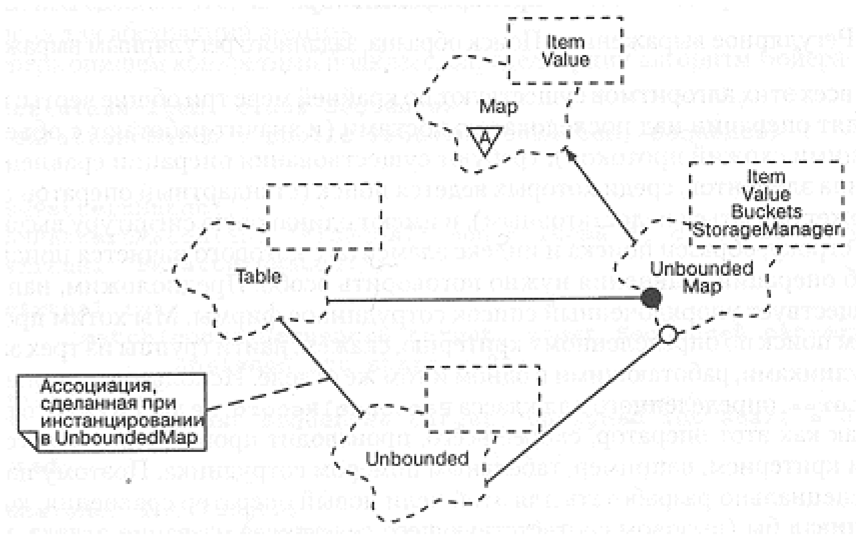


Рис. 9-12. Классы поддержки

Инструменты

Для нашей библиотеки основная роль шаблонов заключается в параметризации структур типами элементов, которые будут в них содержаться; поэтому такие структуры называют классами-контейнерами. Но, как видно из определения класса **Table**, шаблоны можно использовать также для передачи классу некоторой информации о реализации.

Еще более сложная ситуация возникает при создании инструментов, которые оперируют с другими структурами. Как уже отмечалось, алгоритмы тоже можно представить в виде классов, объекты которых будут выступать в роли агентов, ответственных за выполнение алгоритма. Такой подход соответствует идее Джекоб-сона об объекте управления, который служит связующим звеном, осуществляющим взаимодействие обычных объектов [16]. Преимущество данного подхода состоит в возможности создания семейств алгоритмов, объединенных наследованием. Это не только упрощает их использование, но также позволяет объединить концептуально схожие алгоритмы.

Рассмотрим в качестве примера алгоритмы поиска образца внутри последовательности. Существует целый ряд подобных алгоритмов:

• Простой Поиск образца последовательной проверкой всей структуры. В худшем случае временной показатель сложности данного алгоритма будет 0(рп), где р - длина образца и н- длина последовательности.

• Кнут-Моррис-Пратт Поиск образца с временным показателем 0(р+п). (Knuth-Morris-Pratt) Алгоритм не требует создания копий, поэтому годится для поиска в потоках.

• Бойер-Мур Поиск с сублинейным временным показателем (Boyere-Mooге) 0(с(р+п)), где с меньше 1 и обратно пропорционально р.

• Регулярное выражение Поиск образца, заданного регулярным выражением.

У всех этих алгоритмов существуют по крайней мере три общие черты: все они проводят операции над последовательностями (и значит работают с объектами, имеющими схожий протокол), требуют существования операции сравнения для того типа элементов, среди которых ведется поиск (стандартный оператор сравнения может оказаться недостаточным), и имеют одинаковую сигнатуру вызова (целевую строку, образец поиска и индекс элемента, с которого начнется поиск).

Об операции сравнения нужно поговорить особо. Предположим, например, что существует упорядоченный список сотрудников фирмы. Мы хотим произвести в нем поиск по определенному критерию, скажем, найти группы из трех записей с сотрудниками, работающими в одном и том же отделе. Использование оператора **operator==**, определенного для класса **PersormelRecord**, не даст нужного результата, так как этот оператор, скорее всего, производит проверку в соответствии с другим критерием, например, табельным номером сотрудника. Поэтому нам придется специально разработать для этой цели новый оператор сравнения, который запрашивал бы (вызовом соответствующего селектора) название отдела, в котором работает сотрудник. Поскольку каждый агент, выполняющий поиск по образцу, требует своей функции проверки на равенство, мы можем разработать общий протокол введения такой функции в качестве части некоторого абстрактного базового класса. Рассмотрим в качестве примера следующее объявление:

```
template<class Item, class Sequence> class PatternMatch
{ public:
    PatternMatch();
    PatternMatch(int ( *isEqual)(const Item& x, const Item&
y));
    virtual ~PatternMatch();
    virtual void setIsEqualFunction(int (*)(const Item& x,
const Item& y));
    virtual int
    match(const Sequence& target, const Sequences; pattern,
unsigned int start = 0) = 0;
    virtual int
    match(const Sequence&; target, unsigned int start = 0) =
O;
    protected:
    Sequence rep;
    int (*isEqual)(const Item& x, const Item& y);
    private:
    void operator=(coust PattemMatcb&) {} void
operator==(const PatternMatch&) {} void operator!=(const
PatternMatch&) {}
    };
```

Операции присваивания и сравнения на равенство для объектов данного класса и его подклассов невозможны, поскольку мы использовали соответствующие идиомы. Мы сделали это, потому что операции присваивания и сравнения не имеют смысла для абстракций агентов.

Теперь опишем конкретный подкласс, определяющий алгоритм Бойера-Мура:

```
template<class Item, class Sequence>
class BMPatternMatch ; public PatternMatch<ltem, Sequence> {
public:
BMPatternMatch();
BMPattemMatch(int (*isEqual) (const Item& x, const Item& y));
virtual ~BMPattemMatch();
virtual int
match(const Sequence& target, const Seque unsigned int start = 0);
virtual int
match(const Sequence& target, unsigned in
Protected:
unsigned int length;
unsigned int* skipTable;
```

Рис. 9-13. Классы поиска

```
void preprogress(const Sequence& pattern);
unsigned int itemsSkip(const Sequence& pattern,const Item& item);
};
```

Открытый протокол этого класса полностью копирует соответствующий протокол своего суперкласса. Кроме того, его описание дополнительно включает два элемента данных и две вспомогательные функции. Одна из особенностей данного класса состоит в создании временной таблицы, которая используется для пропуска длинных неподходящих последовательностей. Эти добавочные элементы нужны для реализации алгоритма.

На рис. 9-13 приведена иерархия классов поиска. Иерархия подобного типа применима для большинства инструментов библиотеки. При этом формируются сходные по структуре семейства классов, что позволяет пользователям легко в них ориентироваться и выбирать те, которые наилучшим образом подходят для их приложений.

9.4. Сопровождение

Одно из наиболее интересных свойств сред разработки заключается в том, что, в случае удачной реализации, они стремятся набрать некую критическую массу функциональности и адаптируемости. Другими словами, если мы правильно выбрали основные абстракции и наделили библиотеку рядом хорошо взаимодействующих между собой механизмов, то вскоре

обнаружим, что клиенты используют наш продукт для решения тех задач, о которых разработчики среды и не подозревали. После того, как определились основные схемы использования среды, имеет смысл сделать их формальной частью самой библиотеки. Признаком правильности конструкции среды разработки является возможность внедрения новых моделей поведения с помощью повторного использования уже существующих свойств продукта и без нарушения принципов его архитектуры.

Одной из таких задач является проблема времени жизни объектов. Может встретиться клиент, который не хочет или не нуждается в использовании полно масштабной объектно-ориентированной базы данных, а планирует лишь время от времени сохранять состояние таких структур, как очереди и множества, чтобы иметь возможность получить их состояние при следующем вызове из той же программы или из другого приложения. Принимая во внимание то, что подобные требования могут возникать довольно часто, имеет смысл дополнить нашу библиотеку простым механизмом сохранения объектов.

Сделаем два допущения, касающихся этого механизма. Во-первых, клиент должен обеспечить потоки, в которые объекты будут записываться и считываться. Во-вторых, клиент обязан обеспечить объектам поведение, необходимое для направления в поток.

Для создания такого механизма есть два альтернативных подхода. Можно построить класс-примесь, обеспечивающий семантику "долгожития"; именно такой подход реализован во многих объектно-ориентированных базах данных. В качестве альтернативы можно создать класс, экземпляры которого выступают в качестве агентов, ответственных за перенаправление различных структур в поток. Для того, чтобы обосновать наш выбор, попробуем оценить преимущества и недостатки того и другого подхода.

Как оказалось, для выбранного очень простого механизма сохраняемости примесь не совсем подходит (зато она очень хорошо вписывается в архитектуру настоящей объектно-ориентированной базы данных). При использовании примеси пользователь должен сам добавить ее к своему классу, зачастую переопределив при этом некоторые служебные функции класса-примеси. В нашем случае, для такого простого механизма это окажется неэффективным, так как пользователю будет легче разработать свои средства, чем дорабатывать библиотечные. Таким образом, мы склоняемся ко второму решению, которое потребует от пользователя лишь создания экземпляра уже существующего класса.

Рис. 9-14 иллюстрирует работу такого механизма, продлевающего жизнь объектов за счет работы отдельного агента. Класс **persist** является дружественным классу **Queue**; мы определяем эту связь внутри описания класса **Queue** следующим образом:

```
        friend class Persist<ltem, Queue<Item>  >;
```

В этом случае классы становятся дружественными только в момент инстан-цирования класса **Queue**. Внедрив подобные описания дружественности в каждый абстрактный базовый класс, мы обеспечиваем возможность использования **Persist** с любой структурой библиотеки.

Параметризованный класс **persist** содержит операции записи и считывания **put** и **get**, а также функции для подключения потоков обмена данными. Мы можем определить данную абстракцию следующим образом:

```
        template<class Item, class Structure> class Persist {
Public:
        Persist();
        Persist(iostream& input, iostream& output);
        virtual ~Persist();
        virtual void setInputStream(iostream&);
        virtual void setOutputStream(iostream&);
```

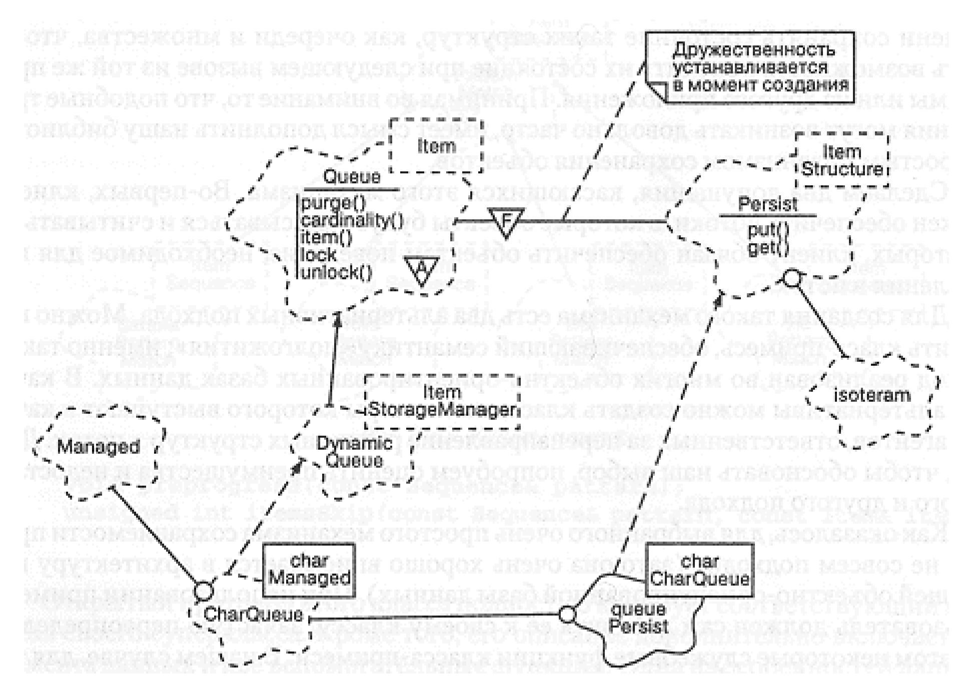


Рис. 9-14. Обеспечение сохраняемости с помощью агента

```
virtual void put (Structure&);
virtual void get (Structure&);
protected:
iostream* inStreain;
iostream* outStream;
};
```

Реализация данного класса зависит от того, является ли он дружественным классу Structure, который фигурирует в качестве аргумента шаблона. В частности, **Persist** зависит от наличия в структуре вспомогательных функций purge, **cardinality**, **itemAt**, **lock**, и **unlock**. Далее срабатывает однородность нашей библиотеки: поскольку каждый базовый класс Structure имеет подобные функции, то **persist** можно безо всяких изменений использовать для работы со всеми имеющимися в библиотеке структурами.

Рассмотрим в качестве примера реализацию функции **Persist::put**:

```
template<class Item, class Structure>
void Persist<Item, Structure>:put(Structure& s)
{
s.lock() ;
unsigned int count = s.cardinality();
(*outStream) " count " endl;
for (unsigned int index = 0; index < count; index++)

(*outStream)      " s.itemAt(index);
s.unlock() ;
}
```

Эта операция использует разработанный нами ранее механизм блокировки, поэтому она будет работать и для защищенных, и для синхронизированных форм. Алгоритм работы функции несложен: сначала в поток выводится количество элементов структуры, а затем, последовательно, все ее элементы. Реализация **persist**:: **get** аналогично выполняет обратное действие:

```
template<class Item, class Structure> void Persist<ltem, Structure>::get(Structure& s)
{
s.lock() ;
```

```
unsigned int count;
Item item;
if (! inStream->eof()) {
(*inStream) " count;
s.purge() ;
for (unsigned int index = 0; (index < count) && (!
inStream->eof()); index++) { (*inStream) " item;
s.add(item);
} } s.unlock () ;
}
```

Для того, чтобы использовать этот простой механизм сохранения данных, клиенту надо всего лишь инстанцировать один дополнительный класс для каждой структуры.

Задача построения среды разработки является довольно сложной. При конструировании основных иерархий классов необходимо учитывать различные, зачастую противоречивые требования к системе. Старайтесь сделать вашу библиотеку как можно более гибкой: никогда нельзя предсказать, как именно попытается ее использовать разработчик. Также очень важно сделать ее как можно более независимой от программной среды - так легче будет использовать ее совместно с другими библиотеками. Предлагаемые абстракции должны быть как можно более простыми, эффективными и понятными разработчику. Самые элегантные решения никогда не будут использованы, если сроки их освоения превысят время, необходимое программисту для решения проблемы своими силами. Сказать, что эффект достигнут, можно будет только когда станет видно, что ваши абстракции используются повторно много раз. То есть, когда разработчик ощутил преимущества их использования и не изобретает велосипед, а сосредоточивает внимание на тех особенностях задачи, которые еще никем не были решены.

Дополнительная литература

Бигерстафф и Перлис (Biggerstaffand Perlis) [H 1989] провели исчерпывающий анализ повторного использования программного обеспечения. Вирфс-Брок (Wirfs-Brock) [C 1988] предложил хорошее введение в объектно-ориентированные среды разработки. Джонсон (Johnson) [G 1992] изучал вопросы документирования архитектуры сред разработки и выявил ряд общих моментов.

Библиотека MacApp [G 1989] для Macintosh является хорошим примером правильно сконструированной объектно-ориентированной прикладной среды разработки. Введение в более раннюю версию этой библиотеки классов может быть найдено у Шмукера (Schmucker) [G 1986]. В недавней работе Голдстейн и Алджер (Goldstein and Alger) [C 1992] обсуждают развитие объектно-ориентированного программного обеспечения для Macintosh.

Другие примеры сред разработки: гипермедиа (Мейровиц (Meirowitz) [C 1986]), распознавание образов (Йошида (Yoshida) [C 1988]), интерактивная графика (Янг (Young) [C 1987]), настольные издательские системы (Феррел (Ferrel) [K 1989]). Среды разработки общего назначения: ET++ (Вейнанд (Weinand) [K 1989]) и управляемые событиями MVC-архитектуры (Шэн (Shan) [G 1989]). Коггинс (Coggins) [C 1990] изучил, в частности, развитие библиотек для C++.

Эмпирическое изучение объектно-ориентированных архитектур и их влияния на повторное использование можно найти в работе Льюиса (Lewis) [C 1992].

Глава 10

Архитектура клиент-сервер: складской учет

Создание большинства бизнес-приложений требует решения целого комплекса задач по хранению данных, обеспечению параллельного доступа к ним, их целостности и защиты. Для этой цели обычно используются готовые системы управления базами данных (СУБД).

Конечно, любая СУБД требует адаптации к условиям конкретного предприятия, которую организации часто разбивают на две задачи: проектирование данных поручается специалистам по базам данных, а программная поддержка выполнения транзакций - программистам. Реализация такого подхода, имеющего, конечно, свои преимущества, сопряжена с решением ряда серьезных проблем. Надо откровенно признать, что в деятельности разработчиков баз данных и программистов существуют серьезные различия, которые определяются различиями в технологии и в навыках разработки. Проектировщики баз данных обычно описывают проблемную область в терминах "долгоживущих" монолитных таблиц с информацией, в то время как программисты привыкли воспринимать мир в терминах потоков управления.

Если эти два подхода не удастся совместить в рамках одного проекта, то добиться целостности проектного решения для более или менее сложной системы будет практически невозможно. Для системы, в которой главное -данные, мы должны добиться равновесия между базой данных и приложением. База данных, разработанная без учета того, как она в дальнейшем будет использоваться, оказывается, как правило, неуклюжей и неэффективной. В свою очередь изолированное приложение может предъявить невыполнимые требования к базе данных, что приведет к серьезным проблемам с обеспечением целостности информации.

Еще в недалеком прошлом бизнес-приложения выполнялись на больших ЭВМ, что воздвигало для обычного служащего почти непреодолимые барьеры на пути к нужной ему информации. Однако с пришествием персонального компьютера ситуация резко переменилась: доступные инструменты обработки и хранения данных вкупе с компьютерными сетями позволили соединить компьютеры не только внутри офиса, но и между предприятиями, отделенными друг от друга тысячами километров. Одним из основных факторов, способствовавших такому изменению, было внедрение архитектуры клиент-сервер. Как отмечает Мимно, "Резкий переход к архитектурам клиент-сервер на базе персональных компьютеров был вызван прежде всего требованиями бизнеса. Перед лицом возросшей конкуренции и ускорившегося цикла выпуска новой продукции, возникла потребность в более быстром продвижении товаров на рынок, увеличении объема услуг, предоставляемых клиентам, более оперативном отслеживании тенденций развития рынка, общем уменьшении расходов" [1]. В этой главе мы рассмотрим пример информационно-управляющей системы (MIS, management information system) и покажем, как объектно-ориентированная технология предлагает единую концепцию организации базы данных и разработки соответствующего приложения для архитектуры клиент-сервер.

10.1. Анализ

Определение границ задачи

Требования к системе складского учета показаны на врезке. Это достаточно сложная программная система, затрагивающая все аспекты, связанные с движением товара на склад и со склада. Для хранения продукции служит, естественно, реальный склад, однако именно программа является его душой, без которой он потеряет свою функцию эффективного центра распределения.

При разработке такой системы заказчикам необходимо частично переосмыслить весь бизнес-процесс и учесть уже имеющиеся программы,

чтобы не потерять вложенные средства (см. главу 7). И хотя некоторое улучшение производительности ведения дел в компании может быть достигнуто просто за счет автоматизации уже существующей системы учета товаров "вручную", радикального улучшения можно добиться только при кардинальном пересмотре ведения бизнеса. Вопросы реинжениринга связаны с системным планированием и выходят за рамки нашей книги. Однако, так же как архитектура системы определяет ее реализацию, общее видение бизнеса определяет всю систему. Исходя из данной предпосылки, начинать следует с рассмотрения общего плана ведения складского учета. По результатам системного анализа можно выделить семь основных функций системы:

• Учет заказов Прием заказов от клиентов и ответы на запросы клиентов о состоянии заказов.

• Ведение счетов Направление счетов клиентам и отслеживание платежей. Прием счетов от поставщиков и отслеживание платежей поставщикам.

• Отгрузка со склада Составление спецификаций на комплектацию товаров, отправляемых со склада клиентам.

Требования к системе складского учета

В качестве части стратегии по проникновению компании, занимающейся торговлей по каталогам, на новые участки рынка, было решено создать ряд относительно автономных региональных складов продукции. Каждый такой склад несет ответственность за учет товаров и выполнение заказов. В целях повышения эффективности своей работы склад обязан сам поддерживать ту номенклатуру товаров, которая в наилучшей степени соответствует потребностям местного рынка. Номенклатура, таким образом, может быть разной для каждого региона. Кроме того, номенклатура должна оперативно меняться в соответствии с изменяющимися потребностями клиентов. Головная компания хотела бы иметь на всех складах одинаковые системы учета. Основными функциями системы являются:

• Учет товаров, приходящих от разных поставщиков, при их приеме на склад.

• Учет заказов по мере их поступления из центральной удаленной организации; заказы также могут приниматься по почте. Их обработка ведется на местах.

• Генерация указаний персоналу, в частности, об упаковке товаров.

• Генерация счетов и отслеживание оплат.

• Генерация запросов о поставке и отслеживание платежей поставщикам.

Кроме автоматизации стандартных складских операций, система также должна предоставлять богатые возможности по генерации различных форм отчетности, в том числе отражающих тенденции развития рынка, списков наиболее надежных и ненадежных поставщиков и клиентов, материалов для рекламных компаний.

• Складской учет Постановка прибывающих товаров на учет и снятие товаров с учета при отправке заказов.

• Закупки Заказ товаров поставщикам и отслеживание поставок.

• Получение Принятие на склад товаров от поставщиков.

• Планирование Выпуск отчетов, в том числе отражающих тенденции спроса на отдельные виды товаров и активность поставщиков.

Не удивительно, что системная архитектура будет отражать перечисленные функциональные свойства. На рис-10-1 приведена диаграмма, иллюстрирующая состав вычислительных элементов сети. Эта структура

является типичной для большинства информационных управляющих систем: группы персональных компьютеров передают информацию на центральный сервер баз данных, который служит центральным хранилищем всех существенных для предприятия данных.

Рассмотрим некоторые детали структуры сети. Во-первых, хотя на рисунке мы видим, что каждый компьютер принадлежит только одной функциональной группе, это не означает, что на нем нельзя выполнять другие операции: отдел бухгалтерского учета должен, например, иметь возможность осуществлять общие запросы к базе данных, а отдел закупок, в свою очередь, просматривать бухгалтерскую информацию, касающуюся платежей поставщикам. Кроме того, руководство компании может в соответствии со своими соображениями перераспределять компьютерные

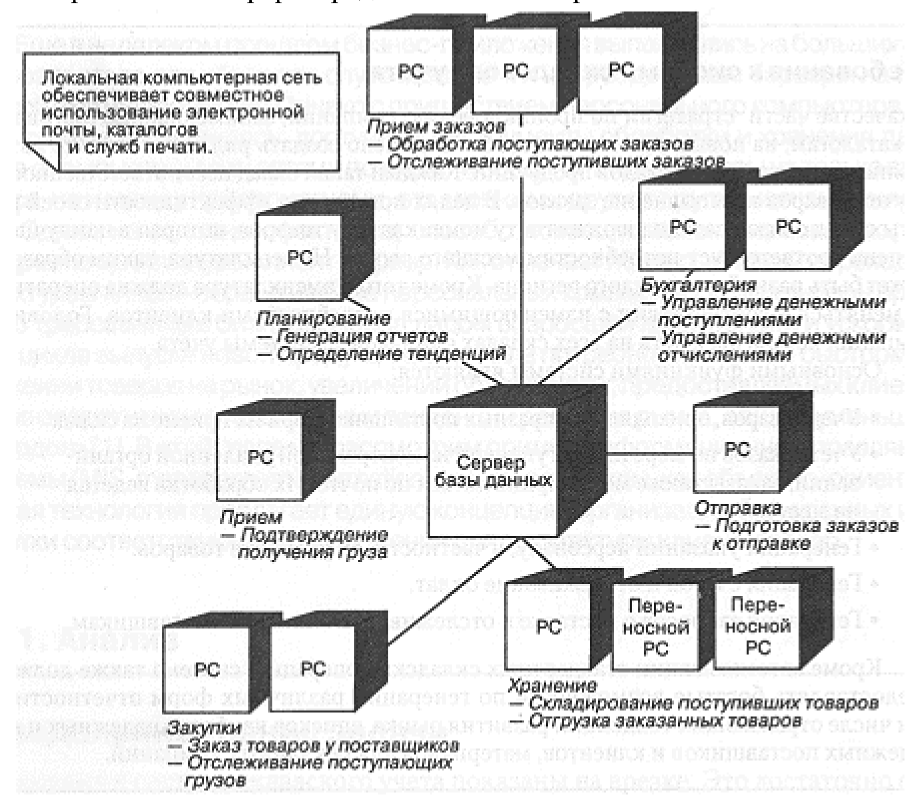


Рис. 10-1. Сеть системы складского учета

ресурсы между отделами фирмы. Требования по защите информации накладывают, однако, некоторые ограничения на доступ к ней: кладовщик, например, не должен иметь возможность отсылать платежные документы. Контроль за доступом к данным обычно осуществляется с помощью общих для всей системы механизмов защиты данных.

Мы предполагаем существование компьютерной сети (LAN), связывающей все компьютеры и обеспечивающей работу общих механизмов взаимодействия пользователей: электронной почты, разделенного доступа к каталогам, вывода информации на сетевой принтер, коммуникаций. Для нашей системы складского учета выбор сетевой операционной среды не так уж и важен, лишь бы она надежно и эффективно обеспечивала взаимодействие пользователей.

Присутствие в нашей схеме переносных персональных компьютеров отражает возможности передовых коммуникационных технологий беспроводной связи. Такой техникой планируется оснастить кладовщиков. По мере поступления новых товаров на склад они будут оперативно заносить в компьютер информацию о его количестве и местоположении на складе, и передавать ее непосредственно на сервер. При необходимости отгрузки товара со склада информация о его количестве и расположении будет сообщаться кладовщику, который передаст ее грузчикам.

Упомянутые технологические средства не сравнимы с космическими станциями, - вся аппаратная часть является стандартной. Что касается программной части, то мы надеемся, что значительная ее часть также будет составлена из стандартных компонентов. Для решения многих локальных подзадач выгодно приобрести готовые электронные таблицы, бухгалтерские пакеты и средства групповой работы. Однако основной движущей силой системы должна стать программа складского учета, связывающая в единое целое все ее составные части.

В подобных приложениях собственно вычисления занимают очень мало места. Основная задача состоит в обеспечении хранения, доступа и пересылки больших объемов данных. Таким образом, большинство архитектурных решений будет нацелено на работу с декларативной информацией (какие товары присутствуют на складе, что скрывается под их обозначением, где они расположены), а не с процедурными вопросами (каким образом идет перемещение товара). Разработка проекта будет основываться на основном принципе объектно-ориентированного подхода:

выявление ключевых абстракций, формирующих словарь предметной области, и механизмов, оперирующих данными абстракциями.

Бизнес-процесс ставит перед нашей системой важное условие: она должна быть открытой для дальнейших модификаций. В ходе анализа нам необходимо выделить ключевые абстракции, играющие в данный момент важную роль в деятельности фирмы: определить типы хранимых в базе данных; перечислить отчеты, которые должны генерироваться; научиться обрабатывать запросы и проводить все остальные транзакции, необходимые в деятельности компании. Именно в данный момент, так как бизнес подвержен постоянным изменениям, фирма все время ищет новые области вложения капитала, и информационная система должна легко перестраиваться в соответствии с модернизацией стратегии фирмы и/или с ее выходом на новые рынки. Устаревшая программная система может стать причиной неудач в бизнесе и вызвать непроизводительную трату людских ресурсов. Таким образом, при проектировании системы складского учета необходимо предусмотреть возможность внесения в нее последующих изменений. Опыт показывает, что наиболее подвержены изменениям следующие элементы программы:

• виды хранимых данных

• аппаратная часть системы.

Каждый из складов с течением времени меняет ассортимент хранимых товаров, начинает работать с новыми клиентами и поставщиками, иногда теряя при этом старых. Может неожиданно оказаться, что о клиентах необходимо хранить дополнительную, не предусмотренную системой информацию.[28] Кроме того, аппаратные технологии все еще развиваются быстрее программных, и компьютеры за несколько лет морально устаревают. Никто, однако, не в состоянии часто менять большие и сложные программные комплексы, это не рационально и непозволительно потому, что время и затраты на создание новой системы часто превосходят время и затраты на покупку и установку новых компьютеров. Внедряя новую систему только по той причине, что старая выглядит устаревшей, вы рискуете своим бизнесом: стабильность и надежность работы являются необходимым свойством программного обеспечения, которое должно обслуживать повседневную деятельность фирмы.

Один из выводов, таким образом, заключается в том, что с течением времени можно ожидать смены интерфейса пользователя. В прошлом бизнес-приложения имели обычный текстовый интерфейс, и это считалось нормальным. Однако общее снижение цен на компьютеры и широкое распространение графических интерфейсов пользователя обуславливают необходимость внедрения графических приложений. Надо помнить, что для системы складского учета интерфейс пользователя является всего лишь

небольшой (хотя и важной) частью. Ядром системы является база данных; пользовательский интерфейс можно рассматривать как оболочку вокруг этого ядра. Для данной системы можно (и даже желательно) создать несколько интерфейсов. Простой, базирующийся на меню, - для клиентов, заполняющих заявки на товар. Современный, типа Windows, - для решения бухгалтерских задач, а также планирования и закупок. Отчеты могут печататься в пакетном режиме, однако некоторым менеджерам могут понадобиться средства для просмотра графиков на экране. Кладовщику нужен простой интерфейс: окна и мышь не подходят для работы в заводских условиях. Мы не будем подробно останавливаться на вопросах, связанных с интерфейсом пользователя; в системе складского учета может быть реализован практически любой из существующих интерфейсов, и это не скажется на ее архитектуре.

Перед дальнейшим обсуждением задачи отметим две важные вещи. Во-первых, при разработке будет использоваться стандартная реляционная база данных (СУРБД), вокруг которой строится программное приложение. Заниматься созданием своей СУБД в данной ситуации просто бессмысленно; нам придется реализовать большинство основных свойств стандартной базы данных, что резко увеличит расходы, а полученный в результате продукт окажется функционально куда менее гибким. Преимущество стандартной реляционной СУБД заключается также в ее относительной переносимости. Большинство распространенных баз данных адаптировано к различным платформам, от персональных компьютеров до мэйнфреймов. Во-вторых, как видно из рис. 10-1, мы хотим, чтобы система складского учета функционировала в распределенной компьютерной сети. Мы планируем разместить всю базу данных на одном компьютере, к которому будут иметь доступ все компьютеры сети. Такая схема и реализует архитектуру клиент-сервер; компьютер, на котором установлена база, является сервером. К нему подключаются несколько клиентов. Конкретный компьютер, на котором работает пользователь, не имеет для сервера никакого значения. Таким образом, наше приложение должно работать на любом компьютере сети, и внедрение новых аппаратных технологий будет оказывать минимальное влияние на функционирование системы.

Архитектура клиент-сервер

Хотя данный раздел и не является подробным обзором архитектуры клиент-сервер, некоторые замечания по этой теме необходимо сделать, так как они напрямую относятся к выбору архитектурных решений для нашей системы.

Что можно отнести к категории клиент-сервер, а что нет, до сих пор является предметом жарких дискуссий.[29] В нашем случае будет достаточно определения решений на базе клиент-сервер как "децентрализованной архитектуры, позволяющей конечным пользователям получать гарантированный доступ к информации в разнородной аппаратной и программной среде. Приложения клиент-сервер сочетают пользовательский графический интерфейс клиента с реляционной базой данных, расположенной на сервере" [2]. Структура таких приложений подразумевает возможность совместной работы пользователей; при этом ответственность за выполнение тех или иных функций ложится на различные, независимые друг от друга элементы открытой распределенной среды. Берсон далее утверждает, что приложение клиент-сервер обычно можно разделить на четыре компонента:

• Логика представления Часть приложения, обеспечивающая связь с инструментами конечного пользователя. Таким инструментом может быть терминал, считыватель штрих-кодов или переносной компьютер. Включает функции: формирование изображения, ввод и вывод информации, управление окнами, поддержка клавиатуры и мыши.

• Бизнес-логика Часть приложения, использующая информацию, вводимую пользователем, и информацию, содержащуюся в базе данных, для выполнения транзакций, удовлетворяющих бизнес-правилам.

• Логика базы данных Часть приложения, "манипулирующая данными приложения... В реляционной базе данных подобные действия обеспечиваются с помощью языка SQL" (SQL, Structured Query Language, язык структурированных запросов).

• Механизмы обращения "Непосредственная работа с базой данных, к базе данных выполняемая СУБД... В идеальном случае механизмы СУБД прозрачны для бизнес-логики приложения" [3].

Один из основных вопросов при проектировании архитектуры системы состоит в оптимальном распределении узлов обработки в сети. Принятие решений здесь усложняется тем, что инструменты и стандарты для архитектур клиент-сервер обновляются с ошеломляющей быстротой. Архитектор должен разобраться, например, с POSIX (Portable Operating System Interface, интерфейс переносимых операционных систем), OSI (Open Systems Interconnection, связь открытых систем), CORBA (Common Object Request Broker, единый брокер объектных запросов), объектно-ориентированным расширением языка SQL (SQL3), и рядом специальных решений фирм-поставщиков типа OLE (Object Linking and Embedding, связывание и внедрение объектов) фирмы Microsoft.[30]

Но на архитектурные решения оказывает влияние не только обилие стандартов. Имеют значение и такие вопросы, как защита данных, производительность системы и ее объем. Берсон предлагает архитектору несколько основных правил проектирования приложении клиент-сервер:

• Компонент логики представления обычно устанавливается там же, где и терминал ввода-вывода, то есть на компьютере конечного пользователя.

• Учитывая возросшую мощность рабочих станций, а также тот факт, что логика представления установлена на машине клиента, имеет смысл там же разместить и некоторую часть бизнес-логики.

• Если механизмы обращения к базе данных связаны с бизнес-логикой, и если клиенты поддерживают некоторое взаимодействие низкого уровня и квазистатические данные, то механизмы обращения к базе данных можно также разместить на стороне клиента.

• Принимая во внимание тот факт, что сетевые пользователи обычно организованы в рабочие группы, и что рабочая группа совместно использует базу данных, фрагменты бизнес-логики и механизмов обращения к базе данных, которые являются общими, и сама СУБД должны находиться на сервере [4].

Если нам удастся выбрать верные архитектурные решения и успешно реализовать их тактические детали, модель клиент-сервер даст системе целый ряд преимуществ. Берсон особо выделяет, что архитектура клиент-сервер:

• Позволяет более эффективно использовать новые компьютерные технологии автоматизации.

• Позволяет перенести обработку данных ближе к клиенту, что снижает загрузку сети и уменьшает продолжительность транзакций.

• Облегчает использование графических интерфейсов пользователя, которые стали доступны на мощных современных рабочих станциях.

• Облегчает переход к открытым системам [5]. Надо выделить, однако, следующие моменты риска:

• Если значительная часть логики приложения окажется вынесенной на сервер, то последний может стать узким местом системы, замедляющим работу пользователей (как это часто бывало при использовании мэйнфреймов в архитектуре хозяин-раб).

• Распределенные приложения ... сложнее нераспределенных [б].

Мы уменьшим этот риск, используя объектно-ориентированный подход к разработке.

Сценарии работы

Сейчас, когда мы представили себе систему в целом, продолжим наш анализ и изучим несколько сценариев ее работы. Сначала перечислим ряд основных режимов использования:

• Клиент звонит по телефону в удаленную телемаркетинговую организацию, чтобы сделать заказ.

• Клиент посылает заказ по почте.

• Клиент звонит, чтобы узнать состояние дел по его заказу.

• Клиент звонит, чтобы добавить или убрать некоторые позиции из заказа.

• Кладовщик получает указание отгрузить клиенту необходимое количество товара.

• Служба доставки получает со склада заказанные клиентом товары и готовит их к отправке.

• Бухгалтерия готовит счет для клиента.

• Отдел закупок готовит заказ на новый товар.

• Отдел закупок добавляет или удаляет имя поставщика из списка.

• Отдел закупок запрашивает поставщика о состоянии заказа.

• Отдел приема товара принимает груз от поставщика и проверяет его соответствие заказу.

• Кладовщик заносит новый товар в список.

• Бухгалтерия отмечает прибытие нового товара.

• Плановый отдел генерирует отчет о показателях продаж по различным типам продуктов.

• Плановый отдел генерирует отчет для налоговых органов с указанием количества товаров на складах.

Каждый из основных сценариев может включать в себя ряд вторичных:

• Заказанного клиентом товара нет на складе.

• Заказ клиента неверно оформлен, или в нем присутствуют несуществующие или устаревшие идентификаторы товаров.

• Клиент звонит, чтобы проверить состояние заказа, но не помнит точно что, кем и когда было заказано.

• Кладовщик получил расходную накладную, но некоторые перечисленные в ней товары не нашлись.

• Служба доставки получает заказанные клиентом товары, но они не соответствуют заказу.

• Клиент не заплатил по счету.

• Отдел закупок делает новый заказ, но поставщик либо ушел из бизнеса, либо больше не поставляет заказанный тип товара.

• Отдел приема товара принимает груз, не полностью соответствующий заказу.

• Кладовщик хочет разместить на складе новый товар, но обнаруживается, что для него нет места.

• Изменяются налоговые коды, что вынуждает плановый отдел составить новый инвентаризационный список находящихся на складе товаров.

Для системы такой сложности, наверно, будут выявлены десятки основных сценариев и еще большее количество вторичных. Этот этап анализа может занять несколько недель, пока не удастся добиться более или менее подробного уровня детализации.[31] Поэтому мы настоятельно советуем применять правило восьмидесяти процентов: не ждите, пока сформируется полный список всех сценариев (никакого времени на это не хватит), изучите около 80% наиболее интересных из них и, если возможно, попытайтесь хотя

бы оценочно проверить правильность общей концепции. В этой главе мы подробно остановимся на двух основных сценариях.

На рис. 10-2 представлен сценарий, в котором покупатель размещает свой заказ в телемаркетинговой фирме. В выполнении этой системной функции задействовано несколько различных объектов. И хотя управление осуществляется взаимодействием клиента (aCustomer) с агентом (anAgent), есть и другие ключевые объекты, а именно: сведения о клиенте (aCustomerRecord), база данных о товарах (inventoryDatabase) и заявка на комплектование (aPackingOrder), являющиеся

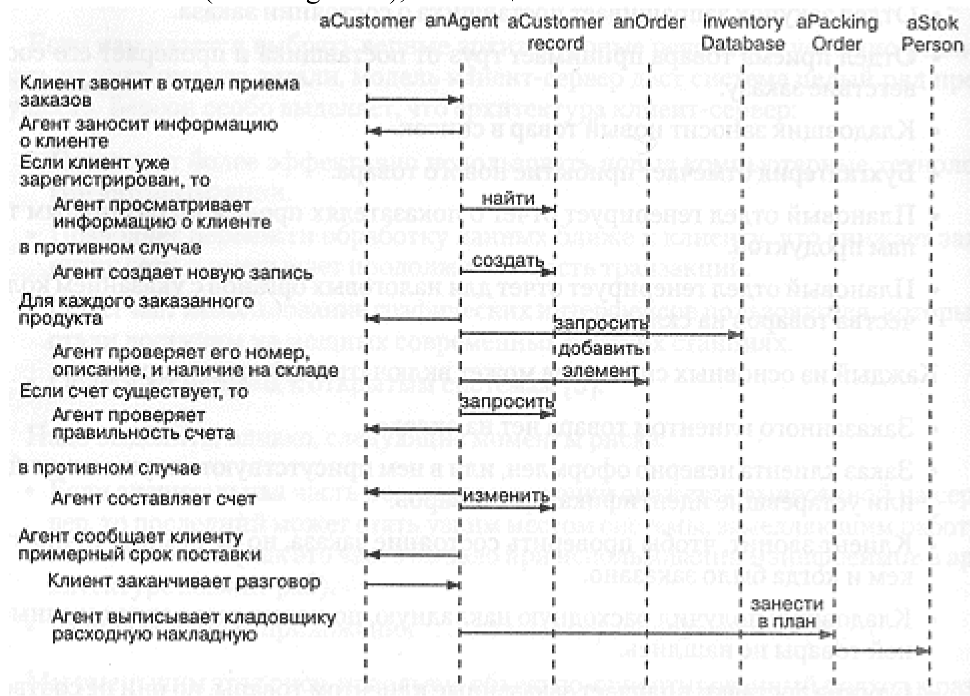


Рис. 10-2. Сценарий заказа

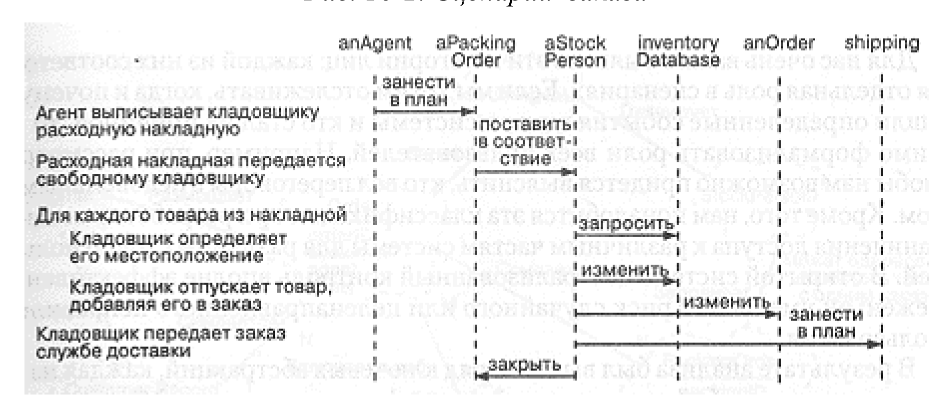


Рис. 10-3. Сценарий выполнения заказа

абстракциями системы складского учета. Этот список абстракций формируется как раз на этапе рассмотрения сценариев работы.

Рис. 10-3 отражает продолжение данного сценария. На нем представлена схема взаимодействия кладовщика и расходной накладной. Мы видим, что здесь кладовщик является главной фигурой. Он взаимодействует с другими объектами, например с отгрузкой (shipping), которой не было в предыдущем сценарии. Однако большинство объектов, фигурирующих на рис. 10-3, присутствуют также и на рис. 10-2, хотя они играют в этих сценариях различные роли. Например, в сценарии взаимодействия с клиентом мы создаем заказ (anOrder) как документ, в котором отражены требования клиента. В складском сценарии тот же самый заказ исполняется.

При составлении каждого из таких сценариев мы должны постоянно задавать себе ряд вопросов. Какой объект будет нести ответственность за выполнение

того или иного действия? Как объект будет проводить ту или иную операцию: самостоятельно или используя свойства другого объекта? Не слишком ли много операций вменяется в круг обязанностей данного объекта? Что произойдет при ошибке в ходе выполнения сценария (какие постусловия могут нарушиться)? Что случится, если будут нарушены некоторые предусловия?

Занимаясь подобным антропоморфизмом для каждого функционального свойства системы, мы откроем в системе целый ряд интересных объектов высокого уровня. Сначала перечислим лиц, взаимодействующих с системой:

- Customer клиент
- Supplier поставщик
- OrderAgent сотрудник отдела продаж
- Accountant бухгалтер
- ShippingAgent сотрудник отдела отгрузки
- Stockperson кладовщик
- PurchasingAgent сотрудник отдела закупок
- ReceivingAgent сотрудник отдела приема товаров
- Planner сотрудник планового отдела

Для нас очень важно выявить эти категории лиц: каждой из них соответствует своя отдельная роль в сценариях. Если мы хотим отслеживать, когда и почему произошли определенные события внутри системы и кто стал их причиной, то необходимо формализовать роли всех пользователей. Например, при рассмотрении жалобы нам возможно придется выяснить, кто вел переговоры с недовольным клиентом. Кроме того, нам понадобится эта классификация при разработке механизма ограничения доступа к различным частям системы для различных групп пользователей. В открытой системе централизованный контроль вполне эффективен и неизбежен: он уменьшает риск случайного или целенаправленного неправильного использования.

В результате анализа был выделен ряд ключевых абстракций, каждая из которых представляет собой определенный тип информации в системе:

- CustomerRecord информация о клиенте
- ProductRecord информация о товаре
- SupplierRecord информация о поставщике
- Order заказ от клиента
- PurchaseOrder заказ поставщику
- Invoice счет
- PackingOrder расходная накладная
- StockingOrder приходная накладная
- ShippingLabel документ на отгрузку

Классы CustomerRecord, ProductRecord И SupplierRecord связаны соответственно с абстракциями Customer, Product и Supplier. Мы, однако разделили эти два типа абстракций, так как они будут играть несколько разные роли.

Заметим, что существуют два вида счетов: те, которые посылаются компанией клиентам для оплаты заказанного товара, и те, которые компания получает от поставщиков товаров. Не отличаясь ничем по своей структуре, они, тем не менее, играют совершенно разные роли в системе.

По классам Packingorder и StockingOrder потребуются некоторые дополнительные разъяснения. В соответствии с первыми двумя сценариями, после того, как сотрудник отдела продаж (OrderAgent) принимает заказ (order) от клиента (Customer), он должен дать указание кладовщику (StockPerson) на выдачу заказанного товара. В нашей системе соответствующая транзакция связана с объектом класса Packingorder (расходная накладная). Этот класс ответственен за сбор всей информации, касающейся выписки расходной

накладной по данному заказу. На операционном уровне это означает, что наша система формирует, а затем передает заказ на переносной компьютер одного из свободных в данный момент кладовщиков. Такая информация должна, как минимум, включать в себя идентификационный номер заказа, наименование и количество каждого из товаров. Нетрудно догадаться, как можно намного улучшить данный сценарий: наша система в состоянии передать кладовщику местоположение товаров, и, возможно, даже примерную последовательность вывоза их со склада, обеспечивающую максимальную эффектив

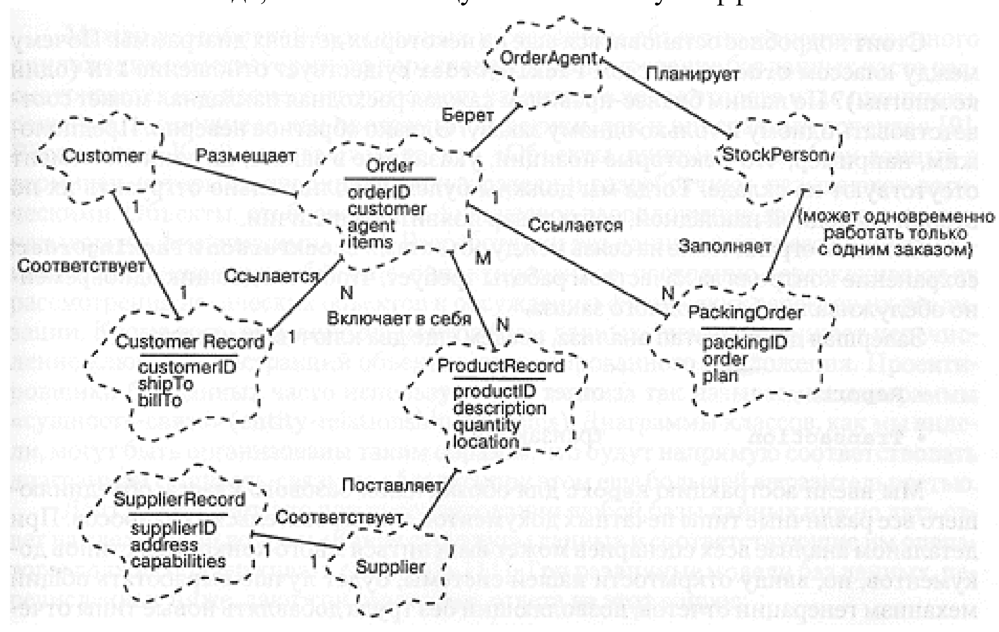


Рис. 10-4. Ключевые классы при приеме и выполнении заказа

ность этой операции. [32] В нашей системе достаточно информации, чтобы обеспечить помощь недавно принятому на работу кладовщику - например, дать ему возможность вывести на экран своего переносного компьютера изображение внешнего вида того или иного товара. Такая поддержка может пригодиться и опытному кладовщику на период смены ассортимента товаров.

Рис. 10-4 содержит диаграмму классов, которая отражает наше понимание процесса взаимодействия некоторых из перечисленных абстракций в сценарии приема и выполнения заказа. Мы дополнили эту диаграмму некоторыми украшениями атрибутов, играющих важную роль в функционировании каждого из классов.

Основные мотивы введения именно такой структуры классов связаны с учетом перехода между экземплярами классов. Получив заказ, мы бы хотели, в частности, сформировать маркер, обозначающий клиента, сделавшего заказ; для этого необходимо перейти от экземпляра класса заказа (order) обратно к клиенту (customer). Получив расходную накладную, надо возвратиться к клиенту и к сотруднику отдела продаж для передачи информации об отгрузке; это означает, что нам потребуется перейти от расходной накладной к заказу, и затем от него - к клиенту и сотруднику отдела продаж. Что касается клиента, то желательно знать, какие товары он чаще всего заказывает в то или иное время года. Для выполнении такого запроса необходимо вернуться от клиента ко всем предыдущим его заказам.

Стоит подробнее остановиться еще на некоторых деталях диаграммы. Почему между классом Order и классом PackingOrder существует отношение 1:N (один ко многим)? По нашим бизнес-правилам каждая расходная накладная может соответствовать одному и только одному заказу. Однако обратное неверно. Предположим, например, что некоторые позиции, указанные в заказе, на данный момент отсутствуют на складе. Тогда мы должны будем дополнительно отгрузить их по второй расходной накладной, когда товар появится в наличии.

Отметим ограничение на связь между объектами StockPerson и PackingOrder:

сохранение контроля за качеством работы требует, чтобы кладовщик одновременно обслуживал не более одного заказа.

Завершая данный этап анализа, введем еще два ключевых класса:

- Report отчет
- Transaction Транзакция

Мы ввели абстракцию Report для обозначения базового класса, объединяющего все различные типы печатных документов и пользовательских запросов, При детальном анализе всех сценариев может выясниться много конкретных типов документов, но, ввиду открытости нашей системы, будет лучше выработать общий механизм генерации отчетов, позволяющий без труда добавлять новые типы отчетов. Действительно, выделив общие для всех отчетов свойства, мы сможем наделить их общим поведением и структурой, что позволит придать соответствующим элементам системы стандартизованный вид и облегчит для конечного пользователя работу с системой.

Наш список далеко не полон, но у нас накопилось достаточно информации для перехода к разработке архитектуры системы. Однако, до того, необходимо рассмотреть некоторые принципы, влияющие на организацию структур данных внутри программы.

Модели баз данных

Дэйт рассматривает базу данных как "вместилище хранимой информации. Она, как правило, одновременно является и интегрированной, и общедоступной. Под "интегрированностью" имеется в виду то, что базу данных можно представить как объединение нескольких отдельных файлов данных, избыточность информации в которых частично или полностью исключена... Под "общедоступностью" имеется в виду то, что информация, содержащаяся в базе, может одновременно использоваться сразу несколькими пользователями" [7]. При централизованном управлении базой данных можно "устранять несоответствия, устанавливать стандарты, накладывать ограничения на доступ к информации и поддерживать целостность базы данных" [8].

Разработка эффективной базы данных является трудной задачей, так как к ней предъявляется много взаимно противоречивых требований. Проектировщик должен учитывать не только функциональные требования к приложению, но также быстродействие и размер базы данных. Базы данных, неэффективные по быстродействию, оказываются, как правило, бесполезными. Системы, для реализации которых надо забить компьютерами все здание и нанять толпу администраторов для ее поддержки, неэффективны с точки зрения стоимости.

Между разработкой базы данных и созданием объектно-ориентированного приложения существует много параллелей. Проектирование баз данных часто рассматривается как процесс итеративного развития, в ходе которого надо принимать решения, касающиеся как программной логики, так и аппаратных аспектов [9]. Вёрковски и Кул указывают на то, что "Объекты, описывающие базу данных в терминах, которыми оперируют пользователи и разработчики, называются логическими. Объекты, отображающие физическое расположение данных в системе, называются физическими" [10]. Разработчики баз данных в процессе проектирования, напоминающем объектно-ориентированное, постоянно перескакивают от рассмотрения логических объектов к обсуждению физических аспектов их реализации. Кроме того, описание элементов базы данных очень напоминает перечисление ключевых абстракций объектно-ориентированного приложения. Проектировщики баз данных часто

используют для анализа так называемые диаграммы "сущность-связь" (entity-relationship diagrams). Диаграммы классов, как мы видели, могут быть организованы таким образом, что будут напрямую соответствовать диаграммам сущность-связь, но обладать при этом еще большей выразительностью.

Дэйт утверждает, что при проектировании любой базы данных нужно дать ответ на следующий вопрос: "Какие структуры данных и соответствующие им операторы должна поддерживать система?" [11]. Три различные модели баз данных, перечисленные ниже, дают три различных ответа на этот вопрос:

- иерархическая
- сетевая
- реляционная.

Недавно появился четвертый тип, а именно объектно-ориентированные базы данных (ООСУБД). ООСУБД соединяют традиционную технологию проектирования баз данных с объектной моделью. Применение такого подхода оказалось достаточно полезным в таких областях, как компьютерное проектирование (CAE) и разработка программ с помощью компьютеров (CASE), где нам приходится манипулировать значительными объемами данных с разнообразным семантическим содержанием. Объектно-ориентированные базы данных могут дать для некоторых приложений значительный выигрыш в быстродействии по сравнению с традиционными реляционными базами данных. В частности, в случае наличия большого количества связей между таблицами, объектно-ориентированные базы данных могут работать значительно быстрее, чем реляционные. Более того, ООСУБД гарантируют согласованную "бесшовную" интеграцию данных и бизнес-правил. Чтобы достичь той же семантики, в реляционных базах используют сложную систему триггеров, которые формируются с помощью языков программирования третьего и четвертого поколений - модель, которую никак нельзя назвать ясной и понятной.

Однако по ряду причин многие компании считают, что использование реляционной базы данных в контексте объектно-ориентированной архитектуры менее рискованно. Технология реляционных баз данных значительно более зрелая, она реализована на широком спектре различных платформ и зачастую предлагает более полный набор средств защиты, контроля версий и поддержания целостности. Кроме того, компания, уже вложившая определенный капитал в кадры и в инструменты, поддерживающие реляционную модель, просто не может позволить себе изменить за одну ночь всю технологию работы.

Реляционная модель весьма популярна. Принимая во внимание ее большую распространенность, широкий набор программных продуктов, ее поддерживающих, а также тот факт, что она удовлетворяет функциональным требованиям к системе складского учета, мы выбрали именно ее. Таким образом, мы остановились на гибридном решении: построение объектно-ориентированной оболочки над традиционной реляционной базой и использование преимуществ обоих подходов. Рассмотрим вкратце некоторые основные принципы проектирования реляционных баз данных. Зная их, мы лучше поймем, как создать объектно-ориентированную оболочку.

Основными элементами реляционной базы данных являются "таблицы, в которых столбцы представляют собой предметы и их атрибуты, а строки описывают отдельные экземпляры предметов... Модель также подразумевает наличие операторов для генерации новых таблиц на базе старых: именно таким способом пользователи могут манипулировать данными и получать информацию из базы" [12].

Рассмотрим для примера базу данных склада с радиоэлектронными товарами, на котором хранятся резисторы, конденсаторы и микросхемы. Каждый тип продукции в соответствии с предыдущей диаграммой классов

обладает уникальным идентификационным номером и описательным именем. Например:

products

productId	description
0081735	Resistor, 10 ае 1/4 watt
0081736	Resistor, 10 ае 1/4 watt
3891043	Capacitor, 100 pF
9074000	7400 1C quad NAND
9074001	74LS00 1C quad HAND

Мы видим таблицу с двумя столбцами, каждый из которых представляет определенный атрибут. В данном случае порядок, в котором расположены строки (столбцы), не важен; количество строк не ограничено, но каждая из них должна быть уникальной. Первый столбец, productID, является первичным ключом, то есть он может быть использован для однозначной идентификации детали.

Товары поступают от поставщиков; информация о каждом из них должна содержать уникальный идентификатор поставщика, имя компании, ее адрес, и, возможно, телефонный номер. Таким образом, можно составить следующую таблицу:

Suppliers

SupplierID	Company	Address	Telephone
00056	Interstate Supply	2222 Fannin, Amarillo, TX	806-555-0036
03107	Interstate Supply	3320 Scott, Santa Clara, CA	408-555-3600
78829	Universal Products	2171 Parfet Ct, Lakewood, CD	303-555-2405

supplierID - первичный ключ в том смысле, что им можно однозначно идентифицировать поставщика. Отметим, что все строки в этой таблице уникальны, однако у двух из них имя поставщика одинаково.

Различные поставщики предлагают различные продукты по различным ценам, поэтому мы можем организовать также таблицу стоимости продуктов. Она содержит текущую цену для каждой комбинации товар/поставщик:

Prices

productID	SupplierID	Price
0081735	03107	$0.10
0081735	78829	$0.09
0156999	78829	$367.75
7775098	03107	$10.69
6889655	00056	$0.09
9074001	03107	$1.75

В этой таблице нет простого первичного ключа. Для однозначной идентификации строк мы должны использовать комбинацию ключей productID и supplierID. Ключ, образуемый из значений различных столбцов, называется составным. Заметьте, что мы не включили в эту таблицу названия деталей и поставщиков - это было бы излишним; данную информацию можно отыскать по значениям полей productID и supplierID в таблицах товаров и поставщиков. Поля productID и supplierID называются внешними ключами, так как они представляют первичные ключи других таблиц.

На рис. 10-5 представлена структура классов, соответствующая этим таблицам. Здесь, для обозначения записей, которые имеют смысл только в совокупности с записями из других таблиц, мы используем ассоциацию с атрибутом. Первичные ключи таблиц заключены в квадратные скобки.

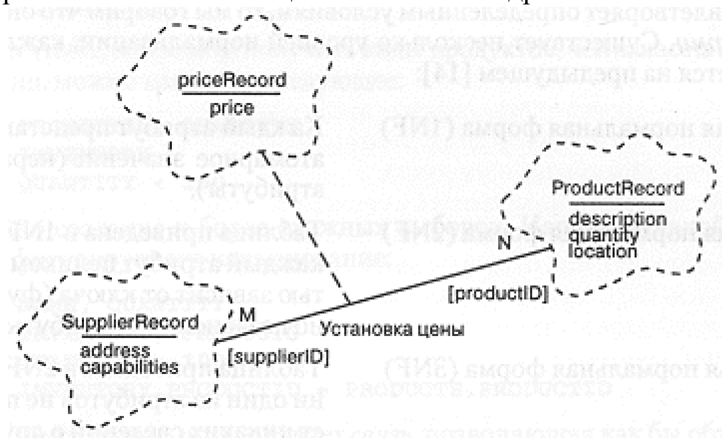


Рис. 10-5. Ассоциация с атрибутами

Далее, мы можем проверить состояние склада с помощью таблицы, содержащей количество всех имеющихся в наличии продуктов:

Inventory

ProductId	Quantity
0081735	1000
0097890	2000
0156999	34
7775098	46
6889655	1
9074001	92

Эта таблица показывает, что объектно-ориентированное представление данных системы может отличаться от их представления в базе данных. В схеме, представленной на рис. 10-4, quantity является атрибутом класса ProductRecord, а здесь, в целях обеспечения быстродействия, мы решили разместить quantity в отдельной таблице. Дело в том, что, как правило, описание товара (description) модифицируется очень редко, в то время как количество (quantity) меняется постоянно по мере того, как со склада отгружаются товары и на склад прибывают новые грузы. Для оптимизации доступа к количеству товара разумнее выделить его в отдельную таблицу.

Данная деталь реализации системы, как следует из рис. 10-4, не будет видна клиентам нашего приложения. Класс productRecord создает иллюзию того, что quantity является его частью.

Самой очевидной и в то же время наиболее важной целью проектирования базы данных является построение такой схемы размещения данных, при которой каждый факт хранится в одном и только в одном месте. При этом не происходит дублирования информации, упрощается процесс внесения изменений в базу данных и поддержания ее целостности (согласованности и правильности данных). Достигнуть цели не всегда бывает легко (оказывается, что это и не всегда нужно). Тем не менее, в нашем случае данное свойство будет очень желательным.

Для достижения этой цели (важной, но не единственной [13]) была разработана специальная теория нормализации. Нормализация есть свойство таблицы; если таблица удовлетворяет определенным условиям, то мы говорим что она имеет нормальную форму. Существует несколько уровней нормализации, каждый из которых базируется на предыдущем [14]:

• Первая нормальная форма (1NF)   Каждый атрибут представляет собой атомарное значение (неразложимые атрибуты).

• Вторая нормальная форма (2NF) Таблица приведена в 1NF, и при этом каждый атрибут целиком и полностью зависит от ключа (функционально независимые атрибуты).

• Третья нормальная форма (3NF) Таблица приведена в 2NF, и при этом ни один из атрибутов не предоставляет никаких сведений о другом атрибуте (взаимно независимые атрибуты).

Таблица в третьей нормальной форме "содержит свойства ключа, весь ключ и ничего кроме ключа" [15].

Все рассмотренные таблицы находятся в 3NF. Существуют еще более высокие уровни нормализации, в основном связанные с многозначными фактами, но в данном случае они не имеют для нас большого значения.

Для того, чтобы связать воедино объектно-ориентированную схему и реляционную модель, иногда нам приходится сознательно нарушать нормализацию таблиц, внося в них определенную избыточность. При этом нам придется прилагать специальные усилия для поддержания синхронизации избыточных данных, однако взамен мы получаем возможность более быстрого доступа к данным, что для нас важнее.

SQL

При работе с объектно-ориентированной моделью, где данные и формы поведения соединены воедино, пользователю может понадобиться осуществить ряд транзакций с таблицами. Он, например, может захотеть добавить в базу нового поставщика, исключить из нее некоторые товары или изменить количество имеющегося в наличии товара. Может также появиться необходимость сделать различные выборки из базы данных, например, просмотреть список всех продуктов от определенного поставщика или получить список товаров, количество которых на складе недостаточно или избыточно с точки зрения заданного нами критерия. Может, наконец, понадобиться создать исчерпывающий отчет, в котором оценивается стоимость пополнения запасов до определенного уровня, используя наименее дорогих поставщиков. Подобные типы транзакций присутствуют почти в каждом приложении, использующем реляционную базу данных. Для взаимодействия с реляционными СУБД разработан стандартный язык - SQL (Structured Query Language, язык структурированных запросов). SQL может использоваться и в интерактивном режиме, и для программирования.

Самой важной конструкцией языка SQL является предложение SELECT следующего вида:

SELECT <attribute> FROM <relation> WHERE <condition>

Для того чтобы, например, получить коды продуктов, чей запас на складе меньше 100 единиц, можно написать следующее:

SELECT PRODUCTID, QUANTITY FROM INVENTORY WHERE QUANTITY < 100

Возможно создание и более сложных выборок. Например такой, где вместо кода товара фигурирует его наименование:

SELECT NAME, QUANTITY FROM INVENTORY, PRODUCTS WHERE QUANTITY < 100 AND INVENTORY.PRODUCTID = PRODUCTS.PRODUCTID

В этом предложении присутствует связь, позволяющая как бы объединять несколько отношений в одно. Данное предложение SELECT не создает новой таблицы, но оно возвращает набор строк. Одна выборка может содержать сколь угодно большое число строк, поэтому мы должны иметь средства для доступа к информации в каждой из них. Для этого в языке SQL введено понятие курсора, смысл которого схож с итерацией, о которой мы говорили в главе 3. Можно, например, определить курсор следующим образом:

```
DECLARE C CURSOR
FOR SELECT    NAME, QUANTITY
FROM          INVENTORY,   PRODUCTS
WHERE         QUANTITY <  100
AND           INVENTORY.PRODUCTID - PRODUCTS.PRODUCTID
```

Чтобы открыть эту выборку, мы пишем

OPEN C

Для прочтения записей выборки используется оператор FETCH:

FETCH C INTO NAME, AMOUNT

И, наконец, после того, как работа завершена, мы закрываем курсор;

CLOSE c

Вместо использования курсора можно пойти другим путем: создать виртуальную таблицу, где содержатся результаты выборки. Такая виртуальная таблица называется представлением. С ним можно работать как с настоящей таблицей. Создадим, например, представление, содержащее наименование товара, имя поставщика и стоимость:

```
CREATE VIEW V (NAME, COMPANY, COST)
AS   SELECT    PRODUCTS.NAME, SUPPLIERS.COMPANY,
PRICES.PRICE FROM      PRODUCTS, SUPPLIERS, PRICES WHERE
PRODUCTS.PRODUCTID . PRICES.PRODUCTID AND
SUPPLIERS.SUPPLIERID - PRICES.SUPPLIERID
```

Использование представлений предпочтительнее, так как оно позволяет создавать различные представления для различных клиентов системы. Поскольку представления могут существенно отличаться от низкоуровневых связей в базе данных, гарантируется некоторая степень независимости данных. Права доступа пользователей к информации можно определять на основе виртуальных, а не реальных таблиц, позволяя таким образом записывать безопасные транзакции. Представления несколько отличаются от таблиц, хотя бы тем, что связи в представлениях не могут быть обновлены напрямую.

В нашей системе SQL-эапросы будут играть роль абстракций низкого уровня. Пользователи вряд ли будут разбираться в SQL, ведь этот язык не является частью предметной области. Мы будем использовать SQL при реализации программы. Составлять свои SQL-предложения смогут только достаточно искушенные в программировании разработчики инструментальных средств нашей системы. От простых смертных, работающих с системой каждый день, язык запросов будет скрыт.

Рассмотрим следующую задачу: получив заказ, мы хотим определить имя сделавшей его компании. С точки зрения программиста SQL, это нетрудная задача. Однако, в нашем случае, когда основное программирование выполняется на C++, мы предпочли бы использовать следующее выражение:

currentOrder.customer().name()

С точки зрения объектно-ориентированного подхода это выражение вызывает селектор customer, возвращающий ссылку на клиента, а затем - селектор name, возвращающий имя клиента. На самом деле данное выражение вычисляется следующим запросом:

```
SELECT    NAME
FROM      ORDERS, CUSTOMERS
WHERE     ORDERS.CUSTOMERID -
CURRENTORDER.CUSTOMERID
AND       ORDERS . CUSTOMERID = CUSTOMERS .CUSTOMERID
```

Спрятав от клиента детали реализации данного вызова, мы скрыли от него все неприятные особенности работы с SQL.

Отображение объектно-ориентированного представления мира в реляционное концептуально ясно, но обычно требует довольно утомительной проработки деталей [33] По замечанию Румбаха, "Соединение объектной модели с реляционной базой данных - в целом довольно простая задача, за исключением вопросов, связанных с обобщением" [16]. Румбах предлагает также некоторые правила, которые следует учитывать при отображении классов и ассоциаций (включая агрегацию) на таблицы:

• Каждый класс отображается в одну или несколько таблиц.

• Каждое отношение "многие ко многим" отображается в отдельную таблицу.

• Каждое отношение "один ко многим" отображается в отдельную таблицу или соотносится с внешним ключом [17].

Далее он предлагает три альтернативных варианта отображения иерархии наследования в таблицы:

• Суперкласс и каждый его подкласс отображаются в таблицу.

• Атрибуты суперкласса реплицируются в каждой таблице (и каждый подкласс отображается в отдельную таблицу).

• Атрибуты всех подклассов переносятся на уровень суперкласса (таким образом мы имеем одну таблицу для всей иерархии наследования) [18].

Нет ничего удивительного в том, что существуют определенные ограничения по использованию SQL в низкоуровневой реализации.[34] В частности, этот язык поддерживает ограниченный набор типов данных, а именно, символы, строки фиксированной длины, целые числа и вещественные числа с фиксированной и плавающей точкой. Отдельные реализации иногда умеют работать и с другими типами данных; однако представление информации в виде графических элементов или строк произвольной длины напрямую не поддерживается.

Анализ схем данных

Дэйт задается следующим вопросом: "Пусть дан набор данных, которые надо расположить в базе данных. Как определить подходящую логическую структуру для этих данных? Другими словами, как определить связи и атрибуты? Это и есть задача проектирования базы данных" [19]. Оказывается, что идентификация ключевых абстракций базы данных во многом напоминает процесс идентификации классов и объектов. По этой причине мы начнем разработку системы складского учета сразу с объектно-ориентированного анализа, в процессе которого будет формироваться структура базы данных, а не будем сперва браться за создание схемы базы данных, и затем выводить из нее объектную модель.

Начнем с уже перечисленного нами списка основных абстракций. Применив к нему правила Румбаха, мы получим следующие таблицы базы данных (сначала перечислим те из них, которые соответствуют ролям групп, принимающих участие в работе системы):

• CustomerTable
• SupplierTable
• OrderAgentTable
• AccountantTable
• ShippingAgentTable
• StockPersonTable
• RecetvingAgentTable
• Flannel-Table

Затем следуют таблицы, отражающие классификацию продуктов и их наличие на складе:

• ProductTable
• InventoryTable

И, наконец, мы вводим таблицы для документопотока:
• OrderTable
• PurchaseOrderTable
• InvoiceTable
• PackingOrderTable
• StockOrderTable
• ShippingLabelTable

Мы не создавали таблиц для классов Report и Transaction, - результаты анализа подсказывают, что объекты этих классов не нуждаются в хранении.

На следующем этапе анализа можно в деталях определить состав атрибутов всех перечисленных таблиц. Наверно, нет смысла обсуждать на страницах этой книги данные вопросы; мы уже останавливались на наиболее интересных свойствах этих абстракции (см. рис. 10-4), а оставшиеся атрибуты дают мало нового с точки зрения архитектуры системы.

10.2. Проектирование

Формулируя подходы к архитектуре системы складского учета, мы должны помнить о трех моментах организационного характера: разделение функций между клиентской и серверной частью, механизм управления транзакциями, стратегия реализации клиентской части приложения.

Архитектура клиент/сервер

Наиболее важным вопросом реализации архитектуры клиент/сервер является не столько вопрос о том, где будет проведена граница между этими двумя частями, сколько о том, как разумно произвести это разделение. Возвращаясь к первоосновам, ответ на этот вопрос нам известен: нужно сосредоточиться на поведении каждой абстракции, основывающемся на анализе вариантов использования каждой сущности, и только затем принять решение о размещении поведения. После того, как мы проделаем такую работу в отношении нескольких основных объектов, станут ясны общие механизмы, понимание которых поможет нам правильно разместить оставшиеся абстракции.

Для примера рассмотрим поведение классов Order и ProductRecord. Анализ первого из них дает нам следующий перечень необходимых операций:
• construct
• setCustomer
• setOrderAgent
• addItem
• removeItem
• orderID
• customer
• orderAgent
• numberOfItems
• itemAt
• quantityOf
• totalValue

Перечисленные сервисные операции можно сразу выразить на языке C++, предварительно дав два новых определения типов:

// типы идентификационных номеров typedef unsigned int OrderID;

// тип, описывающий местную валюту typedef float Money;

Теперь получаем следующее определение класса:

class Order { public:

Order();

```
Order(OrderID);
Orderfconst Order&);
-Order() ;
Orders operator=(const Orders);
int operator==(const Orders) const;
int operator!=(const Orders) const;
void setCustomer(Customer&);
void setOrderAgent(OrderAgent&);
void addItem(Product&, unsigned int quantity = 1);
void removeItem(unsigned int index, unsigned int quantity = 1);
OrderID orderID() const;
Customer& customer() const;
OrderAgent& orderAgent() const;
unsigned int numberOfItem() const;
Product& itemAt (unsigned int) const;
unsigned int quantityOf(unsigned int) const;
Money totalValue() const;
protected:
...
};
```

Обратим внимание на наличие нескольких вариантов конструктора. Первый из них используется по умолчанию (Order ()) для создания объекта с новым уникальным значением идентификатора OrderID. Копирующий конструктор также создает объект с уникальным идентификатором, но при этом копирует в него состояние объекта, использованного в качестве аргумента.

Последний конструктор принимает в качестве аргумента OrderID, то есть конструирует объект уже существующий в базе данных и извлекает из базы его параметры. Другими словами, в этом случае мы повторно материализуем объект, существующий в базе данных. Такая операция, безусловно, требует выполнения некоторых действий: при восстановлении объекта из базы данных соответствующий SQL-механизм должен либо сделать объект разделяемым, либо синхронизировать состояние двух объектов, созданных в разных приложениях. Детали, конечно, скрыты в реализации и недоступны клиенту, который использует объект, применяя обычный объектный интерфейс.

Реализация описанного подхода не вызывает особых затруднений. Если класс order спроектирован так, что его состояние полностью определяется идентификатором OrderID, то реализация операций сводится к обычным операторам чтения и записи из базы данных. Копии объектов синхронизируются, поскольку соответствующая таблица в базе служит единым репозиторием состояния для всех представлений одного объекта.

Диаграмма объектов на рис. 10-6 иллюстрирует описанный SQL-механизм на примере сценария выставления счета. В сценарии реализованы следующие события:

* aClient активизирует операцию setCustomer применительно к объекту класса Order; объект класса Customer передается в качестве параметра.

• Объект класса Order вызывает селектор customerID с параметром заказчика, позволяющим получить из базы данных соответствующий первичный ключ.

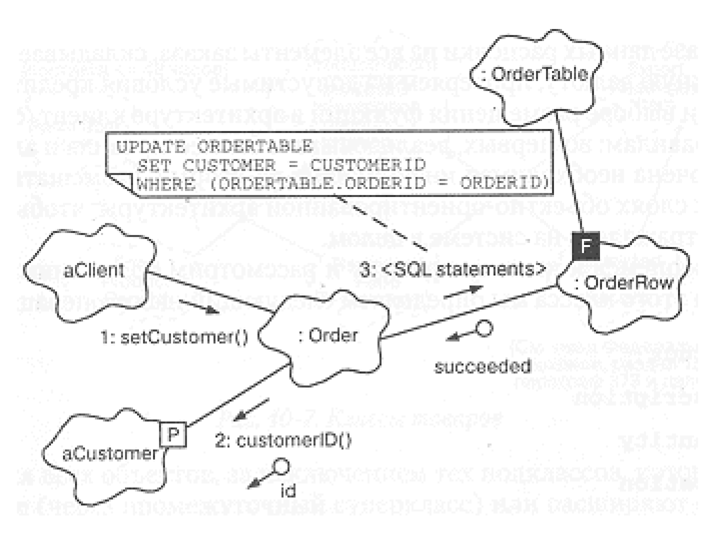


Рис. 10-6. Выставление счета

• Объект, соответствующий заказу, использует SQL-оператор UPDATE, чтобы установить идентификатор заказчика в базе данных заказов.

Описанный механизм предполагает, что мы можем положиться на существующий в базе данных механизм блокировки записей и взаимного исключения при доступе (представьте себе, что могло бы случиться при одновременном обновлении одной записи из двух приложений). Если этот механизм блокировки должен быть видимым для клиента, то можно воспользоваться тем же подходом, который использовался нами при создании библиотеки классов в главе 9. Ниже мы покажем, что механизм выполнения транзакции позволяет модифицировать за один прием несколько записей в базе, обеспечивая тем самым целостность базы данных.

После реализации описанного механизма вопрос о размещении бизнес-логики имеет скорее тактическое значение. В этом смысле ситуация не отличается от той. которую мы имели бы в не объектно-ориентированной архитектуре. Но в объектно-ориентированной архитектуре можно изменить эти решения, скрыв сам факт изменения от клиента. Таким образом, клиента не затрагивают изменения, которые мы делаем по ходу настройки системы.

Для примера рассмотрим два различных случая. Добавление в базу данных записей о наличии продуктов или удаление их, очевидно, должно согласовываться с бизнес-логикой, поэтому кажется естественным разместить бизнес-логику на сервере. Добавление сведении о новых продуктах в базу данных требует точного их определения и однозначной идентификации. Кроме того, эти данные необходимо сделать доступными для всех клиентов, чтобы обновить их кэшированные таблицы. Удаление продукта из базы также требует проверки на наличие заказов по этому продукту и предупреждения соответствующих клиентов.[35]

Напротив, расчет стоимости заказов является более локальной операцией и его лучше выполнять в клиентской части приложения. Выполняя подсчеты, мы запрашиваем в базе данных расцепки на все элементы заказа, складываем их, пересчитываем в нужную валюту, проверяем на допустимые условия кредитования и т. д.

Итак, при выборе размещения функции в архитектуре клиент/сервер мы следуем двум правилам: во-первых, реализовывать бизнес-правила и алгоритмы там где сосредоточена необходимая информация; во-вторых, размещать эти алгоритмы в нижних слоях объектно-ориентированной архитектуры, чтобы внесение изменении не отражалось на системе в целом.

Теперь вернемся к нашему примеру и рассмотрим более внимательно класс product. Для этого класса мы определяем следующий набор операций:

• construct
• setDescription

• setQuantity
• setLocation
• setSupplier
• productID
• description
• quantity
• location
• supplier

Эти операции являются общими для всех видов товаров. Однако, анализ частных случаев показывает, что есть продукты, для которых эти характеристики недостаточны. С учетом того, что проектируемая система является открытой, а виды товаров могут быть самыми различными, приведем несколько примеров специфических товаров и их свойств:

• Скоропортящиеся продукты, требующие определенного режима хранения.

• Едкие и токсичные химические вещества, также требующие специального обращения.

• Комплектные товары, которые поставляются в определенных сочетаниях (например, радиопередатчики и приемники) и поэтому взаимозависимы.

• Высокотехнологичные компоненты, поставки которых ограничиваются за-конодательством стран-экспортеров.

Перечисленные примеры наводят на мысль о необходимости создания некоторой иерархии классов товаров. Однако, перечисленные свойства настолько различны, что не образуют никакой иерархии. В данной ситуации более целесообразно воспользоваться примесями, что иллюстрирует и рис. 10-7. Обратим внимание на использование в этой диаграмме украшений ограничения, уточняющих семантику каждой абстракции.

Каков смысл наследования для абстракций, отражающих сущности реляционной базы данных? Очень большой: построение иерархии наследования сопровождается вычленением общих признаков поведения и отображением их в структуре суперклассов. Эти суперклассы будут ответственны за реализацию общего

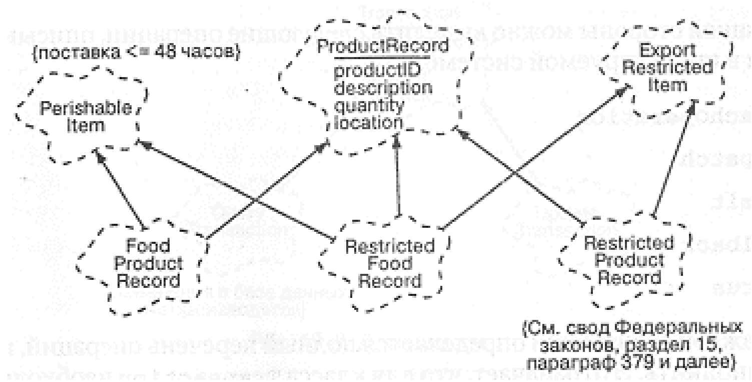


Рис. 10-7. Классы товаров

поведения для всех объектов, за исключением тех подклассов, которые уточняют это поведение (через промежуточный суперкласс) или расширяют его (через суперкласс-примесь). Такой подход не только упрощает построение системы, но и повышает устойчивость к вносимым изменениям за счет сокращения избыточности и локализации общих структур и поведения.

Механизм транзакций

Архитектура клиент/сервер построена на взаимодействии клиентской и серверной частей приложения, для реализации которого необходим

определенный механизм. Берсон указал, что "существует три базовых вида взаимодействия между процессами в архитектуре клиент/сервер" [20]:

- конвейеры (pipes)
- удаленный вызов процедур (RPC)
- взаимодействие клиент/сервер через SQL.

В нашем примере мы воспользовались только третьим способом. Но, в общем случае, могут использоваться все указанные виды взаимодействия в соответствии с требованиями производительности или в результате выбора программных средств конкретного поставщика. В любом случае наш выбор должен быть скрыт, чтобы не оказывать влияния на абстракции высокого уровня.

Мы ранее уже упомянули о классе транзакции, но не остановились подробно на его семантике. Берсон определяет транзакцию как "единицу обмена и обработки информации между локальной и удаленной программами, которая отражает логически законченную операцию или результат" [21]. Это и есть определение нужной нам абстракции: объект-транзакция является агентом, ответственным за выполнение некоторого удаленного действия, а, следовательно, отчетливо отделяет само действие от механизма его реализации.

Действительно, транзакция является основным высокоуровневым видом взаимодействия сервера и клиента, а также между клиентами. На основе этого понятия можно выполнять конкретный анализ вариантов использования. Принципиально все основные функции в системе складского учета могут рассматриваться как транзакции. Например, размещение нового заказа, подтверждение поступления товаров и изменения информации о поставщиках являются системными транзакциями.

С внешней стороны можно выделить следующие операции, описывающие суть поведения в проектируемой системе:

- attachOperation
- dispatch
- commit
- rollback
- status

Для каждой транзакции определяется полный перечень операций, которые она должна выполнить. Это означает, что для класса Transact ion необходимо определить функции-члены, такие как attachoperation, которые предоставляют другим объектам возможность объединить набор SQL-операторов для исполнения в качестве единой транзакции.

Интересно отметить, что такое объектно-ориентированное видение транзакций полностью согласуется с принципами, принятыми в практике работы с базами данных. Дэйт определил, что "транзакция представляет собой последовательность операторов SQL (возможно, не только SQL), которые должны быть неразделимы в смысле произведения отката и управления параллельным доступом".[36]

Концепция атомарности наиболее существенна в семантике транзакций. Если в некоторой транзакции операция выполняется над несколькими строками таблицы, то либо все действия должны быть выполнены, либо содержимое таблицы должно быть оставлено без изменении. Следовательно, когда мы посылаем транзакцию (dispatch), мы имеем в виду выполнение группы операций как единого целого.

При благополучном завершении транзакции мы должны зафиксировать ее результаты (commit). Невыполнение транзакции может произойти в силу ряда причин, в том числе из-за отказов сети или блокировки информации другими клиентами. В таких ситуациях выполняется откат в исходное состояние (rollback). Селектор status возвращает значение параметра, определяющего успешность транзакции.

Выполнение транзакции несколько усложняется при работе с распределенными базами данных. Как реализовать протокол завершения транзакций при работе с локальной базой достаточно понятно, а что необходимо сделать при работе с данными, размещенными на нескольких серверах? Для этого используется так называемый двухфазный протокол завершения транзакций [23]. В этом случае агент, то есть объект класса Transaction, разделяет транзакцию на несколько фрагментов и раздает их для выполнения различным серверам. Это называется фазой подготовки. Когда все серверы сообщили о том, что готовы к завершению, центральный агент транзакции передает им всем команду commit. Это называется фазой завершения. Только при правильном завершении всех разделенных компонент транзакции основная транзакция считается завершенной. Если хотя бы на одном сервере выполнение операций будет неполным, мы откатим всю транзакцию. Это возможно потому, что каждый экземпляр Transaction знает, как откатить свою транзакцию.

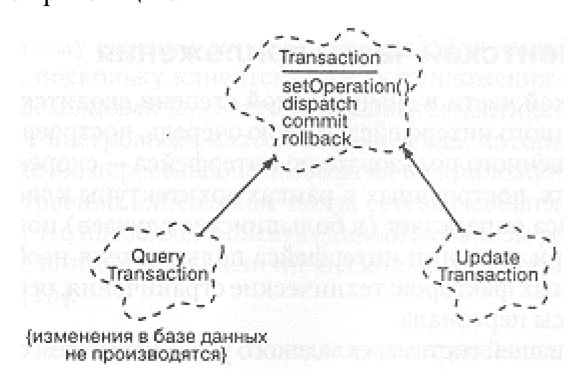


Рис. 10-8. Транзакции

Изложенное выше представление о классе транзакций показано на рис. 10-8. Мы видим здесь иерархию транзакций. Класс Transaction является базовым для всех транзакций и содержит в себе все ключевые аспекты поведения. Производные специализированные классы вносят в общее поведение свои особенности. Мы различаем, например, классы UpdateTransaction и QueryTransaccion, потому что их семантика очень различна: первый из них модифицирует данные на сервере баз данных, а второй - нет. Различая эти и другие типы транзакций, мы собираем в базовом классе наиболее общие характеристики, и пополняем при этом наш словарь.

В процессе дальнейшего проектирования мы, возможно, обнаружим и другие разновидности транзакций, которые будут представлены собственными подклассами. Например, если мы убедимся, что операции добавления и удаления данных из конкретной базы имеют общую семантику, то введем операции AddTransaction и DeleteTransaction, чтобы отразить эту общность поведения.

Во всяком случае, существование базового класса Transaction позволяет выполнять нам любое атомарное действие. Например, на C++ он мог бы выглядеть так:

```
public:
Transaction(};
virtual ~Transact ion ();
virtual void set0peration(const
UnboundedCollection<SQLStatement>&);
virtual int dispatch();
virtual void commit();
virtual void rollbackO;
virtual int status{} const;
protected:
...
```

};

Обратим внимание, что для построения этого класса мы использовали базовые классы, определенные нами в главе 9. В данном случае мы построили транзакцию в форме индексированной коллекции операторов. Для манипулирования этой коллекцией использован параметризованный класс UnboundedCollection.

Принятое архитектурное решение позволяет сложному пользовательскому приложению выполнять наборы SQL-операторов. Все детали реализации механизма управления транзакциями оказываются скрытыми для простых клиентов, которым достаточно выполнять некоторые общие типы транзакции.

Создание клиентской части приложения

Создание клиентской части в значительной степени сводится к построению графического прикладного интерфейса. В свою очередь, построение удобного интуитивного и дружественного пользователю интерфейса - скорее искусство, чем наука. В приложениях, построенных в рамках архитектуры клиент/сервер, именно качество интерфейса определяет (в большинстве случаев) популярность тех или иных программ. При создании интерфейса пользователя необходимо учитывать множество различных факторов: технические ограничения, особенности психологии, традиции, вкусы персонала.

При создании нашей системы складского учета мы можем столкнуться с двумя препятствиями. Во-первых, нужно выяснить, каким должен быть "правильный" интерфейс пользователя. Во-вторых, желательно определить, какие общепринятые подходы мы можем использовать при создании интерфейса.

Ответ на первый вопрос можно получить достаточно просто, но для этого нужно прототипировать, прототипировать и прототипировать. Нужно как можно раньше получить действующую модель системы, чтобы показать ее пользователям и получить от них квалифицированные замечания. Объектно-ориентированный подход существенно поможет нам в этом смысле, поскольку он основан на итерационном развитии проекта. На самых ранних стадиях проекта мы уже сможем показать пользователям прототип системы.

Второй вопрос находится в сфере стратегии проекта, но для его успешного разрешения у нас имеется множество хороших примеров. Существуют коммерческие продукты, например, X Window System от MIT, Open Look, Windows от Microsoft, MacApp от Apple, NextStep от Next, Presentation Manager от IBM. Все эти продукты существенно различаются: некоторые основываются на сети, а некоторые опираются на концепцию ядра, некоторые позволяют действовать на уровне пикселей, а другие считают примитивами более сложные геометрические фигуры. В любом случае все они позволяют существенно упростить создание графического интерфейса пользователя. Ни один из перечисленных продуктов не родился за одну ночь. Все они постепенно развивались из самых простых систем, прошли путь проб и ошибок. В результате эти системы вобрали в себя набор абстракций, достаточный для построения пользовательского интерфейса. Поскольку нет однозначного ответа на вопрос о лучшем интерфейсе, то существуют несколько вариантов оконной модели.

В главе 9 мы уже упоминали о том, что при работе с большими библиотеками классов (каковыми являются и библиотеки графического интерфейса) важно понять механизмы их построения. Для нашей задачи основным механизмом является реакция GUI-приложений на события. Берсон указывал, что для клиентской части приложения существенны события, связанные со следующими объектами [24]:

• мышь

- клавиатура
- меню
- обновление окна
- изменения размера окна
- активизация/деактивация
- начало/завершение.

Мы добавим к этому перечню сетевые события.[37] Для нашей архитектуры они очень существенны, поскольку клиентская часть приложения связана с другими компонентами и приложениями через сеть. Описанная семантика хорошо согласуется с нашим подходом к построению класса Transaction, который может рассматриваться как посредник, пересылающий события от приложения к приложению. С точки зрения построения клиентской части, сетевые события являются разновидностью событий, что позволяет описать единый механизм реакции на события.

Берсон обратил внимание на наличие нескольких альтернативных моделей обработки событий [25]:

• Цикл обработки событий   В цикле просматривается очередь событии и для каждого события вызывается соответствующая процедура обработки.

• Обратный вызов   Приложение регистрирует функцию обратного вызова для каждого элемента GUI; обратный вызов происходит, когда элемент зарегистрирует событие.

• Гибридная модель   Сочетание циклического опроса и функций обратного вызова.

Изрядно упрощая, можно утверждать, что в интерфейсе MacApp используется цикл, в Motif - функции обратного вызова, а Microsoft Windows является примером гибридной модели.

Кроме первичного механизма, нам необходимо реализовать еще множество GUI-механизмов: рисование, прокрутка, работа с мытью, меню, сохранение и восстановление, печать, редактирование, обработка ошибок, распределение памяти. Безусловно, подробное рассмотрение всех этих вопросов находится вне рамок нашего анализа, поскольку каждая конкретная GUI-среда имеет свои собственные реализации этих механизмов.

Мы предлагаем разработчику клиентской части приложения выбрать подходящую GUI-среду разработки, изучить ее основные механизмы и правильно их применить.

10.3. Эволюция

Управление релизами

Теперь, полностью определив архитектурный каркас системы складского учета, мы можем приступить к последовательному развитию. Выберем сначала наиболее важные транзакции в нашей системе (ее вертикальный срез) и выпустим продукт, который по крайней мере симулирует выполнение транзакций.

Для примера остановимся на трех простых транзакциях: занесение в базу нового клиента, добавление товара и принятие заказа. При реализации этих транзакций мы в той или иной степени затронем практически все архитектурные интерфейсы. Если мы сможем успешно преодолеть этот ключевой этап, то дальше будем выпускать релизы в следующем порядке:

• Модификация или удаление данных о клиентах; модификация или удаление данных о продуктах: модификация заказа; запросы о клиентах, заказах и продуктах.

• Интеграция всех похожих транзакции, связанных с поставщиками: создание заказа и выписка счета.

• Интеграция всех оставшихся транзакций, связанных со складом: составление отчетов и выписка расходных накладных.

• Интеграция всех оставшихся транзакций, связанных с бухгалтерией: поступление оплаты.

• Интеграция всех оставшихся транзакции, связанных с отгрузкой.

• Интеграция всех оставшихся транзакций, связанных с планированием.

При общем сроке проектирования системы в 12-18 месяцев необходимо каждые 3 месяца выпускать рабочий релиз программы. К окончанию срока все необходимые для работы системы транзакции будут охвачены.

В главе 6 уже упоминалось, что ключевым моментом при такой стратегам является выявление риска, поэтому для каждого релиза мы находим самое опасное место и активно прорабатываем его. Для приложений клиент/сервер это связано, в первую очередь, с возможно более ранним тестированием вместимости и масшта-бируемости (чтобы как можно раньше найти узкие места системы и сделать с ними что-нибудь). При этом в каждый релиз следует включать транзакции из разных функциональных элементов системы - тогда будет меньше шансов столкнуться с неожиданностями.

Генераторы приложений

При создании приложений типа системы складского учета необходимо произвести множество экранных форм и отчетов. Для больших систем эта работа не столько сложна, сколько велика по объему и однообразна. По этой причине сегодня весьма популярны генераторы приложений на основе языков четвертого поколения (4GL). Использование этих языков не противоречит идеям объектно-ориентированного проектирования. Напротив, 4GL-ЯЗЫКИ позволяют при правильном применении существенно упростить написание кода.

Языки четвертого поколения используются для генерации экранных форм и отчетов. На основании спецификаций они создают исполняемый код форм и отчетов. Мы интегрируем этот код в нашу систему, "оборачивая" его вручную тонким объектно-ориентированным слоем. Таким образом код, сгенерированный 4GL, становится частью структуры классов, которую остальные части приложения могут использовать, не обращая внимание на то, как она была создана.

Такой подход позволяет нам воспользоваться преимуществами 4GL, сохраняя иллюзию полностью объектно-ориентированной архитектуры. Кроме того, языки четвертого поколения сами подвергаются сильному влиянию технологии объект-но-ориентированного программирования и включают в себя прикладные интер-фейсы (API) для объектно-ориентированных языков типа C++.

Такую же стратегию можно использовать и при реализации диалога пользо-вателя с системой. Написание программ для модального и немодального диалог скучно, поскольку мы должны охватить массу мелких деталей. Лучше не писать такой код вручную [38], а использовать GUI-конструкторы, позволяющие "рисовать окна диалога. После получения готового кода мы заворачиваем его в объектную оболочку, включаем в наше приложение и получаем систему с четким разделением обязанностей.

10.4. Сопровождение

Системы клиент/сервер редко бывают окончательно завершенными. Не то чтоб мы никогда не могли сказать про систему, что она уже стабильна.

Просто систем должна развиваться вместе с бизнесом, чтобы оставаться полезной.

Можно указать некоторые направления модернизации, которые вероятны для системы складского учета:

• Предоставить возможность клиентам работать с системой по каналам связи.

• Автоматически генерировать индивидуальные каталоги товаров для потре-бительских групп или даже отдельных клиентов.

• Полностью автоматизировать все функции, устранив кладовщиков и боль-шую часть работающих на приеме и отгрузке.

Анализ показывает, что все перечисленные модификации связаны скорее с со циалъным и политическим риском, чем с техническим. Гибкая объектно-ориенти-рованная архитектура системы позволяет заказчику использовать все степени сво-боды, чтобы адаптироваться к постоянно меняющемуся рынку.

Дополнительная литература

Об архитектуре клиент/сервер написано больше, чем большинство смертных способно прочесть за всю жизнь. Две наиболее полезные ссылки - это Девайр (Dewire) [H 1992] и Берсон (Berson) [H 1992], которые предложили исчерпывающие и хорошо читаемые обзоры по всему спектру проблем технологии клиент/сервер. Блум (Bloom) [H 1993] дал короткое, но интересное перечисление базовых понятий и проблем архитектуры клиент/сервер.

Децентрализация - это не то же самое, что вычисления в архитектуре клиент/сервер, хотя она и предусматривает вычисления в архитектуре клиент/сервер в корпоративных информационно-управляющих системах. Все мотивировки за и против децентрализации можно найти в работе Гвенджерича (Guengerich) [H 1992].

Исчерпывающее обсуждение технологии реляционных баз данных можно найти у Дэйта (Date) [E 1981,1983,1986]. Вдобавок к этому, Дэйт (Date) [E 1987] предложил описание стандарта SQL. Разные подходы к анализу данных могут быть найдены у Вериярда (Veryarcl) [B 1984], Хавришкевича (Hawryszkiewycz) [E 1984) и Росса (Ross) [F 1987).

Объектно-ориентированные базы данных представляют собой сплав обычной технологии баз данных и объектной модели. Отчеты о работе в этой области можно найти у Кэттла (Cattle) (E 1991], Атвуда (Atwood) [E 1991], Дэвиса и др. (Davis et al.) [H 1983], Кима и Ло-човского (Kim and Lochovsky) [H 1989], Здоника и Майера (Zdonik and Maier) [E 1990].

В библиографии приведены несколько ссылок на различные оконные системы и объектно-ориентированные интерфейсы пользователя. Подробности о Microsoft Windows API можно найти в Windows [G 1992], а относительно Apple MacApp - в Macapp [G 1992].

Глава 11

Искусственный интеллект: криптоанализ

Мыслящие существа способны проявлять очень сложные формы поведения, обладая сознанием, механизмы которого мы понимаем очень смутно. Подумайте, например, как вы планируете маршрут поездки по городу, чтобы выполнить массу дел. В плохо освещенном помещении вам удается распознавать границы предметов и избегать столкновений. Вы можете сосредоточиться на беседе с одним собеседником на шумной вечеринке, где много людей говорит одновременно. Ни одна из этих задач не имеет четкого алгоритмического решения. Планирование маршрута относится к классу пр-полных задач. Передвижение в темноте подразумевает принятие решения на основе неполной и нечеткой зрительной информации. Выделение речи одного человека из множества разговоров требует умения улавливать полезную информацию в шуме и отфильтровать нужные сообщения из общей какофонии.

Эти и подобные им проблемы привлекают внимание исследователей в области искусственного интеллекта, которые стремятся улучшить наши представления о разуме человека. В частности, создаются интеллектуальные системы, которые подражают некоторым аспектам поведения человека. Ерман, Ларк и Хайес-Рот указывали, что "интеллектуальные системы отличаются от традиционных рядом признаков (не все из них обяэательны):

• способностью достигать целей, меняющихся во времени;
• способностью усваивать, использовать и преобразовывать знания;
• способностью оперировать с разнообразными подсистемами, варьируя используемые методы;
• интеллектуальным взаимодействием с пользователями и другими системами;
• самостоятельным распределением ресурсов и концентрацией внимания" [1].

Реализация в системе хотя бы одного из этих требований уже является непростой задачей. Еще сложнее сделать интеллектуальную систему для использования в некоторых специфических прикладных областях, например, в медицинской диагностике и диспетчеризации авиарейсов: такие системы должны, как минимум, не причинять вреда, а искусственный интеллект практически ничего не знает о здравом смысле.

Успехи энтузиастов в этой области несколько преувеличены; но, тем не менее, искусственный интеллект дал немало хороших практических идей, в частности представление знаний, концепция информационной доски и экспертные системы [2]. В данной главе рассматриваются подходы к созданию интеллектуальной системы расшифровки криптограмм на основе метода информационной доски, в достаточной степени моделирующего человеческий способ решения задачи. Как мы увидим, методы объектно-ориентированного проектирования очень хорошо работают в этой области.

11.1. Анализ

Определение границ предметной области

Как сказано во врезке, мы намерены заняться криптоанализом - процессом преобразования зашифрованного текста в обычный. В общем случае процесс дешифровки является чрезвычайно сложным и не поддается даже самым мощным научным методам. Существует, например, стандарт шифрования DES (Data Encryption Standard, алгоритм шифрования с закрытым ключом, в котором используются многочисленные подстановки и перестановки), который, по-видимому, свободен от слабых мест и устойчив ко всем известным методам взлома. Но наша задача значительно проще, поскольку мы ограничимся шифрами с одной подстановкой.

В качестве первого шага анализа попробуйте решить (только честно, не заглядывая вперед!) следующую криптограмму записывая, каждый ваш шаг:

Q AZWS DSSC KAS DXZNN DASNN

Подсказка: буква **w** соответствует букве **v** исходного текста. Перебор всех возможных вариантов совершенно лишен смысла. Предполагая, что алфавит содержит 26 прописных английских букв, получим 26! (около 4.03x1026) возможных комбинаций. Следовательно, нужно искать другой метод решения, например, использовать знания о структуре слов и предложений и делать правдоподобные допущения. Как только мы исчерпаем явные решения, мы сделаем наиболее вероятное предположение и будем продвигаться дальше. Если обнаружится, что предположение приводит к противоречию или заводит в тупик, мы вернемся назад и сделаем другую попытку.

Вот наше решение, шаг за шагом:

1. Используя подсказку, заменим **w** на **v**.

Требования к системе криптоанализа

Криптография "изучает методы сокрытия данных от посторонних" [3]. Криптографические алгоритмы преобразовывают сообщения (исходный текст) в зашифрованный текст (криптограмму) и наоборот.

Одним из наиболее общеупотребительных (еще со времен Древнего Рима) криптографических алгоритмов является подстановка. Каждая буква в алфавите исходного текста заменяется другой буквой. Например, можно циклически сдвинуть все буквы алфавита: буква **A** заменяется на **B**, **B** на **C**, а **Z** на **A**. Тогда следующий исходный текст:

CLOS is an object-oriented programming language

превращается в криптограмму:

DMPT jt bo pckfdu-psjfoufe qsphsbnnjoh mbohvbhf

Чаще всего замена делается менее тривиальным образом. Например, **A** заменяется на **G**, **B** на **J** и т. д. Рассмотрим следующую криптограмму:

PDG TBCER CQ TCK AL S NGELCH QZBBR SBAJG

Подсказка: буква **C** в этой криптограмме соответствует букве **O** исходного текста.

Существенно упрощает задачу предположение о том, что для шифрования текста использован алгоритм подстановки, поскольку в общем случае процесс дешифровки не будет столь тривиальным. В процессе расшифровки приходится использовать метод проб и ошибок, когда мы делаем предположение о замене и рассматриваем его следствия. Удобно, например, начать расшифровку с предположения о том, что одно- и двухбуквенные слова в криптограмме соответствуют наиболее употребительным словам английского языка (**I, a, or, it, in, of. on**). Подставляя эти предполагаемые буквы в другие слова, мы можем догадаться о вероятном значении других букв. Например, если трехбуквенное слово начинается с литеры **O**, то это могут быть слова **one,our, off.**

Знание фонетики и грамматики также может способствовать дешифровке. Например, следование подряд двух одинаковых литер с очень малой вероятностью может означать **qq**. Наличие в окончании слова буквы **g** позволяет сделать предположение о наличии суффикса **ing**. На еще более высоком уровне абстракции логично предположить, что словосочетание **it ia** более вероятно, чем **if is**. Необходимо учитывать и структуру предложения: существительные и глаголы. Если выясняется, что в предложении есть глагол, но нет существительного, которое с ним связано, то нужно отвергнуть сделанные ранее предположения н начать поиск заново.

Иногда приходится возвращаться назад, если сделанное предположение вступает в противоречие с другими предположениями. Например, мы допустили, что некоторое двухбуквенное слово соответствует сочетанию **or**, что в дальнейшем привело к противоречию. В этом случае мы должны вернуться назад и попытаться использовать другой вариант расшифровки этого слова, например, **on**.

Требования к нашей системе: по данной криптограмме, в предположении, что использована простая подстановка, найти эту подстановку и (главное) восстановить исходный текст.

Q AZVS DSSC KAS DXZNN DASNN

2. Первое слово из одной буквы, вероятна, **A** или **I**; предположим, что это **A**:

A AZVS DSEC KAS DXZNN DASNN

3. В третьем слове должны быть гласные звуки и вероятно, что это двойные буквы. Это не могут быть **UU** или **II**, а также **AA** (буква **A** уже использована). Попробуем вариант **EE**.

A AZVE DEEC KAE DXZNN DAENN

4. Четвертое слово состоит из трех букв и оканчивается на **E**, это очень похоже на слово **THE**.

A HZVE DEEC THE DXZNN DHENN

5. Во втором слове нужна гласная, и здесь подходят только **I**, **O**, **U** (буква **A** уже использована). Только вариант с буквой **I** дает осмысленное слово.

A HIVE DEEC THE DXINN DHENN

6. Можно найти несколько слов с двойной буквой **E** из четырех букв (**DEER**, **BEER**, **SEEN**). Грамматика требует, чтобы третье слово было глаголом, поэтому остановимся на **SEEN**.

A HIVE SEEN THE SXINN SHENN

7. Смысл в полученном предложении отсутствует, поскольку улей (**HIVE**) не может видеть (**SEEN**), значит, где-то по дороге мы сделали ошибку. Похоже, что выбор гласной буквы во втором слове был неверен, и приходится вернуться назад, отменив самое первое предположение - первым словом должно быть **I**. Повторяя все остальные наши рассуждения практически без изменений мы получаем:

I HAVE SEEN THE SXANN SHENN

8. Посмотрим на два последних слова. Двойная буква **S** в конце не дает осмысленного значения и к тому же уже использована ранее, а вот **LL** дает осмысленное слово.

I HAVE SEEN THE SXALL SHELL

9. Из грамматических соображений очевидно, что оставшееся слово - прилагательное. Анализируя шаблон **S?ALL**, находим **SMALL**.

I HAVE SEEN THE SMALL SHELL

Таким образом, решение найдено. Анализируя процесс решения, мы можем сделать три наблюдения:

- Для решения применялись разнообразные знания: о грамматике, о составе слов, о чередовании согласных и гласных.
- Сделанные предположения регистрировались, потом мы применяли к ним имеющиеся у нас знания и смотрели, что из этого получается.
- Мы подходили к делу наугад, приспосабливаясь к обстановке. Иногда делались выводы от общего к частному (словом из трех букв, оканчивающимся на **E** будет, вероятно, **THE**), а иногда от частного к общему (**?EE?** может соответствовать **DEER**, **BEER**, **SEEN**, но глаголом из них является только **SEEN**).

Изложенный подход известен как метод информационной доски. Он впервые был предложен Ньюэллом в 1962 году, а позднее был использован Редди и Ерманом в проектах Hearsay и Hearsay II по распознаванию речи [4]. Эффективность метода подтвердилась, и он был использован в других областях, включая интерпретацию сигналов, трехмерное моделирование молекулярных структур, распознавание образов и планирование [5]. Метод показал хорошие результаты в представлении описательных знаний; он более эффективен с точки зрения памяти и времени по сравнению с другими подходами [б].

Информационная доска вполне подходит на роль среды разработки (см. главу 9). Попробуем теперь зафиксировать архитектуру этого метода в виде системы классов и механизмов их взаимодействия.

Архитектура метафоры информационной доски

Энглемор и Морган для пояснения модели информационной доски использовали следующую аналогию с группой людей, собирающей фрагменты головоломки в нужную фигуру:

Вообразим себе комнату с большой доской, рядом с которой находится группа людей, держащих в руках фрагменты изображения. Процесс начинают добровольцы, которые размещают на доске наиболее "вероятные" фрагменты изображения (предположим, что они прилепляются к доске). Далее каждый участник группы смотрит на оставшиеся у него фрагменты и решает, есть ли такие, которые подходят к уже находящимся на доске. Участник, нашедший соответствие, подходит к доске и прилепляет свой кусок. В результате фрагмент за фрагментом занимают нужное место. При этом не существенно, что один из участников может иметь больше фрагментов, чем другой. Все изображение будет полностью собрано без всякого обмена информацией между членами группы. Каждый участник активизируется самостоятельно и знает, когда ему нужно включиться в процесс. Никакого порядка подхода к доске заранее не устанавливается. Совместное поведение регулируется только информацией на доске. Наблюдение за процессом демонстрирует его последовательность (по одному фрагменту за подход) и произвольность (когда возникает возможность, фрагмент устанавливается). Это существенно отличается от строгой систематичности, например, от прохождения с левого верхнего угла и перебора каждого фрагмента[7].

Из рис. 11-1 видно, что основу метода составляют три элемента: информаци-онная доска, совокупность источников знаний и управляющий этими источниками контроллер [8]. Отметим, что следующее определение прямо соответствует принципам объектного подхода. Согласно Ни: "Информационная доска нужна для того чтобы хранить данные о ходе и состоянии решаемой задачи, используемые и формируемые источниками знаний. Доска содержит объекты из пространства решений. Эти объекты иерархически группируются по уровням анализа и вместе со своими атрибутами образуют словарь пространства решений" [9].

Энглемор и Морган уточняют: "необходимые для решения задачи знания о предметной области разделены на несколько независимых источников. Каждый источник знаний старается предложить информацию, полезную для решения за дачи. Текущая информация из каждого источника помещается на доске и модифи-

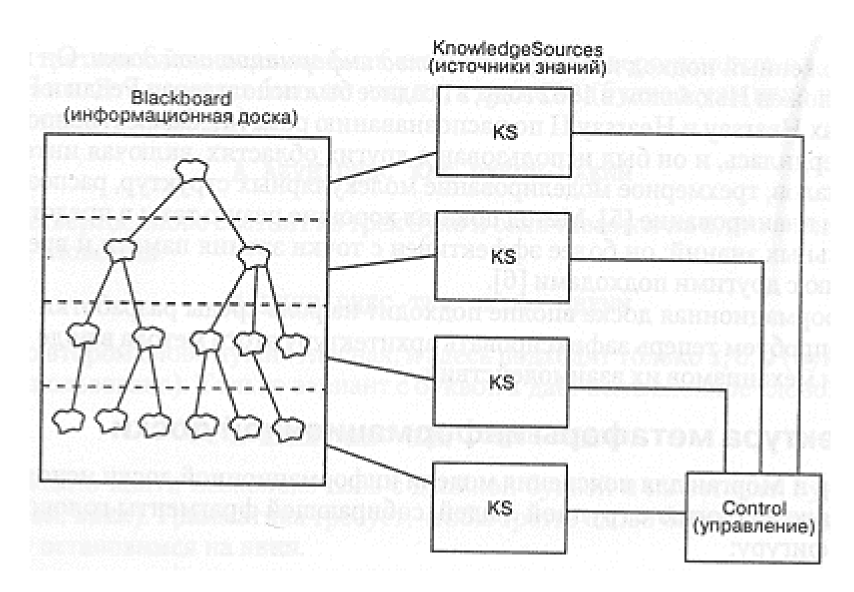


Рис. 11-1. Информационная доска

цируется в соответствии с содержанием знаний. Формой представления источников знаний являются процедуры, наборы правил или логические заключения" [10].

Источники знаний зависят от предметной области. В системах распознавания речи нас могут интересовать агенты, поставляющие знания о фонемах, словах и предложениях. В системах распознавания образов ими могут быть сведения об элементарных структурах изображения, таких, как стыки линий, участки одинаковой плотности, и, на более высоком уровне абстракции, объекты, относящиеся к конкретной сцене (дома, дороги, поля, автомобили и люди).

В общем случае источники знаний соответствуют иерархической структуре объектов, размещаемых на информационной доске. Более того, каждый источник использует объекты одного уровня иерархии в качестве входных данных, а в качестве выхода генерирует или изменяет объекты на другом уровне. Например, в системе распознавания речи источник знаний о словах наблюдает за потоком фонем (низкий уровень абстракции), чтобы обнаружить слово (более высокий уровень абстракции). Источник знаний о предложениях может предположить, что здесь нужен глагол (высокий уровень абстракции) и проверить это предположение, перебрав список возможных слов (низкий уровень абстракции),

Эти два подхода к поиску решения называются соответственно прямой и обратной последовательностью рассуждений. *Прямая последовательность* рассуждений позволяет перейти от более частных предположений к более общим, а *обратная последовательность*, отталкиваясь от некоторой гипотезы, позволяет проверить ее, сравнив с известными предпосылками. Вот почему управление информационной доской мы охарактеризовали как произвольное: в зависимости от обстоятельств, источники знаний могут активизировать либо прямые, либо обратные последовательности рассуждений.

Источники знаний, как правило, состоят из двух компонент: предусловия и действия. Предусловием называется такое состояние информационной доски, которое представляет "интерес" для конкретного источника знаний (потенциально способно его активизировать). Например, в распознавании образов предусловием может быть наличие прямой линии (которая может означать дорогу). Выполнение предусловий заставляет источник знаний сфокусировать внимание на конкретном участке информационной доски, а затем привести в действие соответствующие правила или процедурные знания.

В этих условиях очередность активизации не имеет значения: если источник знаний обнаруживает данные, полезные для решения задачи, он сигнализирует об этом контроллеру доски, фигурально выражаясь, он как бы поднимает руку, показывая, что желает сделать что-то полезное. Из нескольких источников, делающих такой жест, контроллер вызывает того, кто ему представляется наиболее перспективным.

Анализ источников знаний

Вернемся теперь к поставленной задаче и рассмотрим источники знаний, полезные для ее решения. При построении большинства приложений, основанных на знаниях, лучше всего сесть рядом с экспертом в предметной области и фиксировать те эвристики, которые он использует. В нашем случае придется попытаться расшифровать некоторое количество криптограмм и отметить особенности процесса поиска решений.

Действуя таким образом мы выявили тринадцать источников знаний, относящихся к нашей проблеме:

• Префиксы	Наиболее часто используемые начала слов (например, `re`, `anti`, `un`).
• Суффиксы	Наиболее часто используемые окончания слов (`ly`, `ing`, `es`, `ed`).
• Согласные	Буквы, не являющиеся гласными.
• Непосредственно известные подстановки	Подстановки, известные нам априори, до решения задачи.
• Двойные буквы	Наиболее часто сдваиваемые буквы(`tt`,`ll`,`ss`).
• Частота букв	Вероятность появления букв в тексте.
• Правильные строки	Допустимые и недопустимые сочетания букв (например, `qu` и `zg`).
• Сравнение с шаблоном	Слова, соответствующие шаблону.
• Структура фраз	Грамматика, включая знания об именных и глагольных оборотах.
• Короткие слова	Одно-, двух-, трех- и четырехбуквенные слова.
• Решение	Найдено ли решение или мы зашли в тупик.
• Гласные	Буквы, не являющиеся согласными.
• Структура слова	Расположение гласных и типичная структура существительных, глаголов, прилагательных, наречий, предлогов, союзов и т. д.

Исходя из объектно-ориентированного подхода, все эти источники знаний являются потенциальными кандидатами на роль классов, на основе которых создаются объекты, обладающие состоянием (знания), поведением (источник знаний о суффиксах может среагировать на слово с характерным окончанием) и индивидуальностью (знания о коротких словах не зависят от умения сравнивать с шаблоном).

Перечисленные источники знаний можно организовать в иерархию. В частности, существуют группы источников знаний о предложениях, о словах, о группах букв и об отдельных буквах. Такая иерархия соответствует объектам на информационной доске: предложениям, словам, частям слов и буквам.

11.2. Проектирование

Архитектура информационной доски

Теперь у нас есть все, чтобы приступить к решению поставленной задачи с использованием метафоры информационной доски. Это классический пример повторного использования "в большом": мы повторно применяем испытанный архитектурный шаблон как основу проекта. Метод информационной доски предполагает следующие объекты верхнего уровня: информационная доска, несколько источников знаний и контроллер. Остается только определить классы и объекты предметной области, которые специализируют эти общие абстракции.

Объекты информационной доски. Объекты на доске образуют иерархию, отражающую иерархичность различных уровней абстракции источников знаний. Таким образом, у нас есть три следующих класса:

- **`Sentence`** Полная криптограмма
- **`Word`** Отдельное слово в криптограмме
- **`CipherLetter`** Отдельная буква в слове

Источники знаний должны пользоваться общей информацией о сделанных в процессе решения предположениях, поэтому в число объектов информационной доски включается следующий класс:

- **`Assumption`** Предположение, сделанное источником знаний

Наконец, источники знания делают предположения о связи между буквами реального и шифровального алфавитов, так что мы вводим следующий класс:

- **`Alphabet`** Алфавит исходного текста, алфавит криптограммы и соответствие между ними

Есть ли между этими пятью классами что-либо общее? Ответ однозначно утвердительный: все они соответствуют объектам информационной доски и этим существенно отличаются от других объектов, например, источников знаний и контроллера. Поэтому вводится следующий суперкласс для всех ранее перечисленных объектов:

```
class BlackboardObject ...
```

С точки зрения внешнего поведения определим для этого класса две операции:

- **`register`** Добавить объект на доску
- **`resign`** Удалить объект с доски

Почему мы определили эти две операции над объектами класса **`BlackboardObject`**, а не над самой доской? Это похоже на ситуацию, когда объект должен сам нарисовать себя в некотором окне. "Лакмусовый" тест в таких случаях, это вопрос: "Имеет ли сам объект достаточно знаний и умений, чтобы выполнять такие операции?". Объекты информационной доски как раз лучше всех понимают, как им правильно появляться на доске или удаляться с нее (конечно, они нуждаются при этом в помощи самой доски). Мы уже установили ранее, что объекты, взаимодействующие с доской, по своей сути должны самостоятельно включаться в процесс решения задачи.

Зависимости и подтверждения. Предложения, слова и буквы также связаны определенной общностью: для всех них есть соответствующие источники знаний. Конкретный источник знаний, со своей стороны, может проявлять интерес к одному или нескольким таким объектам (зависеть от них) и поэтому фраза, слово и символ шифра должны поддерживать связь с источником знаний, чтобы при появлении предположения относительно объекта уведомлялись соответствующие источники знаний. Это напоминает

механизм зависимостей языка Smalltalk, упомянутый в главе 4. Для реализации этого механизма введем следующий класс-примесь:

```
class Dependent {
public:
      Dependent();
      Dependent(const Dependent&);
      virtual ~Dependent();
      ...
protected
      UnboundedCollection<KnowledgeSource*> references;
};
```

Мы забежали несколько вперед и намекнули на возможную реализацию класса, чтобы показать связь с библиотекой фундаментальных классов, описанной в главе 9. В классе определен один внутренний элемент - коллекция указателей на источники знаний. [39]

Определим для этого класса следующие операции:

- **add** Добавить ссылку на источник знаний
- **remove** Удалить ссылку на источник знаний
- **numberOfDependents** Возвратить число зависящих объектов
- **notify** Известить каждого зависимого

Последняя операция является пассивным итератором: при ее вызове передается как параметр действие, которое надо выполнить над всеми зависящими объектами в коллекции.

Зависимость может примешиваться к другим классам. Например, буква шифра - это объект информационной доски, от которого зависят другие, так что мы можем скомбинировать две этих абстракции для получения нужного поведения. Такое применение примесей поощряет повторное использование и разделение понятий в нашей архитектуре.

Символы шифра и алфавиты имеют еще одно общее свойство: относительно объектов этих классов могут делаться предположения. Вспомните, что предположение (**Assumption**) является одним из объектов на доске (**BlackboardObject**). Так, некоторый источник знаний может допустить, что буква **K** в шифре соответствует букве **P** исходного текста. По мере решения задачи может абсолютно точно выясниться, что **G** означает **J**. Поэтому введен еще один класс:

```
class Affirmation . . .
```

Этот класс отвечает за высказывания (предположения или утверждения) относительно связанного с ним объекта. Мы используем этот класс не как примесь, а для агрегации. Буква, например, не *является* предположением, но может иметь предположение о себе.

В нашей системе предположения допускаются только в отношении отдельных букв и алфавитов. Можно, например, предположить, что какая-либо буква шифра соответствует некоторой букве алфавита. Алфавит состоит из набора букв, относительно которых делаются предположения. Определяя **Affirmation** как независимый класс, мы выражаем в нем сходное поведение этих двух классов, несвязанных наследованием.

Определим следующий набор операций для экземпляров этого класса:

- **make** Сделать высказывание
- **retract** Отменить высказывание
- **chiphertext** Вернуть шифрованный эквивалент для заданной буквы исходного текста
- **plaintext** Вернуть исходный текстовый эквивалент для заданной буквы шифра

Из предыдущего обсуждения видно, что надо ясно различать две роли высказываний: временные предположения о соответствиях между буквами шифра и текста и окончательно доказанные соответствия - утверждена. По мере расшифровки криптограммы может делаться множество различных предположений о соответствии букв шифра и текста, но в конце концов находятся окончательные соответствия для всего алфавита. Чтобы отразить эти роли, уточним ранее выявленный класс **Assumption** в подклассе **Assertion** (утверждение). Экземпляры обоих классов управляются объектами класса **Affirmation** и могут помещаться на доску. Для поддержки введенных ранее операций **make** и **retract** нам необходимо определить следующие селекторы:

- **isPlainLetterAsserted** определена ли эта буква текста достоверно?
- **isCipherLetterAsserted** определена ли эта буква шифра достоверно?
- **plainLetterHasAssumptlon** есть ли предположение об этой букве текста?
- **cipherLetterHasAssumption** есть ли предположение об этой букве шифра?

Теперь мы можем определить класс **Assumption**. Поскольку данная абстракция носит исключительно структурный характер, се состояние можно сделать открытым:

```
class Assumption : public BlackboardObject {
public:
        …
        BlackboardObject* target;
        KnowledgeSource* creator;
        String<char> reason;
        char plainLetter;
        char chipherLetter;
};
```

Отметим, что мы повторно использовали еще один класс среды, описанной в главе 9, а именно, параметризуемый класс **String**.

Класс **Assumption** является объектом информационной доски, поскольку информация о сделанных предположениях используется всеми источниками знаний. Отдельные члены класса выражают следующие его свойства:

- **target** Объект доски, о котором делается предположение
- **creator** Источник знаний, который сделал предположение
- **reason** Основание для сделанного предположения
- **cipherLetter** Предполагаемое значение буквы исходного текста

Необходимость каждого из перечисленных свойств в значительной степени объясняется природой предположений: источник знании формирует предполагаемое соответствие "буква исходного текста - буква шифра" на основании каких-то причин (обычно, некоторого правила). Назначение первого свойства **target** менее очевидно. Оно нужно для отката. Если сделанное предположение не подтвердится, то нужно восстановить состояние объектов на доске, которые воспользовались предположением, а они должны известить источники знаний, что их смысл изменился.

Далее определим подкласс **Assertion**:

```
class Assertion : public Assumption ...
```

Общим для классов **Assumption** и **Assertion** является следующий селектор:

- **isRetractaole** Является ли соответствие потенциально неверным?

Для всех высказанных предположений значение предиката **isRetractable** является истинным, а для утверждений - ложным. Сделанное утверждение уже нельзя ни изменить ни отвергнуть.

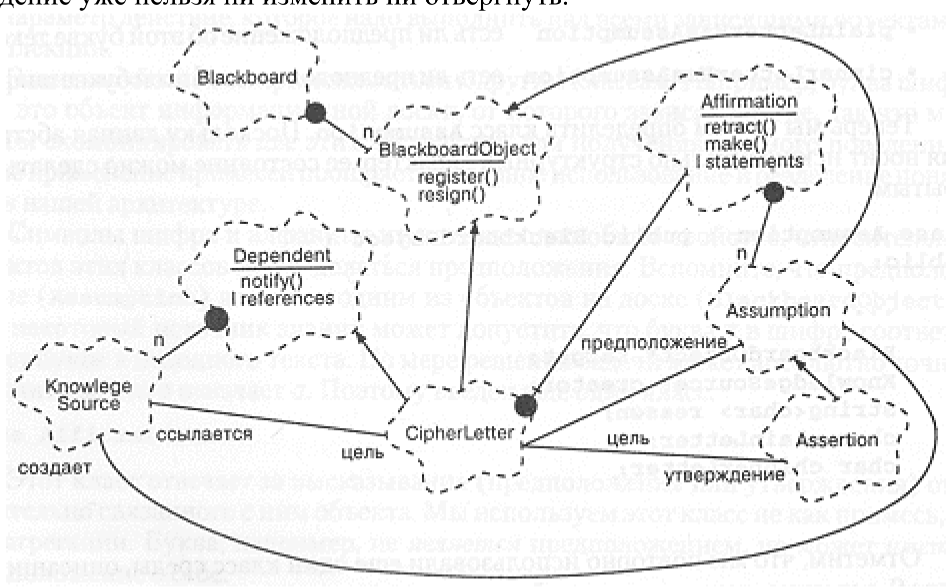


Рис. 11-2. Классы зависимостей и высказываний

На рис. 11-2 приведена диаграмма, поясняющая связь классов зависимостей и высказываний. Обратите особое внимание на роли, которые играют упомянутые абстракции в различных ассоциациях. Например, класс **KnowledgeSource** в одном аспекте является создателем (**creator**) предположения, а в другом - ссылается (**referencer**) на букву шифра. Из различия ролей естественным образом вытекают различия протоколов взаимодействия.

Проектирование объектов информационной доски. Завершим проектирование, добавив кроме класса алфавита классы для предложения (**Sentence**), слова (**Word**) и буквы шифра (**cipherLetter**). Предложение представляет собой просто объект доски (от которого зависят другие объекты), содержащий список слов, Исходя из этого, запишем:

```
class Sentence : public BlackboardObject,
        virtual public Dependent {
public:
        ...
protected:
        List<Word*>. words;
};
```

Суперкласс **Dependent** определен виртуальным, поскольку мы ожидаем, что будут подклассы от **sentence**, которые захотят наследовать также и от **Dependent**. При этом для всех таких подклассов члены класса **Dependent** будут общими.

В дополнение к операциям **register** и **resign** (определенным в суперклассе **BlackboardObject**) и четырем операциям, унаследованным от класса **Dependent**, мы добавляем еще две специфические операции для предложения:

- **value** Текущее значение предложения.
- **isSolved** Истинно, если о всех словах в предложении сделаны утверждения.

Первоначальное значение **value** совпадает с текстом криптограммы. Когда **isSolved** станет истиной, **value** вернет исходный расшифрованный текст.

Слово является объектом доски и источником зависимости. Оно состоит из букв. Для удобства источников знаний в класс слова введены указатели на все предложение, а также на предыдущее и следующее слова в предложении. Описание класса **Word** выглядит так:

```
class Word : public BlackboardObject,
	virtual public Dependent {
public:
	…
	Sentence& sentence() const;
	Word* previous 0 const;
	Word* next() const;
protected:
	List<CipherLetter*> letters;
};
```

Так же как для предложения, в класс слова введены две дополнительные операции:

- **value** Текущее значение слова.
- **isSolved** Истинно, если о всех буквах слова сделаны утверждения.

Теперь можно определить класс **cipherLetter** (буква шифра). Буквы шифра являются объектами информационной доски и порождают зависимости. Кроме того, они имеют значение (буква, как она записывается в шифровке, например, н) и коллекцию возможных предположений и утверждений о соотнесении ее с буквами исходного текста. Для организации коллекции мы используем класс **Affirmation**. Опишем класс буквы следующим образом:

```
class CipherLetter : public BlackboardObject,
	virtual public Dependent {
public:
	…
	char value() const;
	int isSolvedO const; ...
protected:
	char letter;
	Affirmation affirmations;
};
```

Отметим, что и в этот класс добавлена та же пара селекторов по аналогии с классами слова и предложения. Для клиентов этого объекта нужно предусмотреть защищенные операции доступа к предположениям и утверждениям.

Объект **affirmation**s, включенный в этот класс, содержит коллекцию предположений и утверждений в порядке их выдвижения. Последний элемент коллекции содержит текущее предположение или утверждение. Смысл хранения последовательности решения задачи состоит в возможности обучения источников знании на собственных ошибках. Поэтому в класс **Affirmation** введены два дополнительных селектора:

- **moatRecent** возвращает последнее предположение или утверждение
- **statementAt** возвращает n-ое высказывание (предположение или утверждение)

Уточнив поведение класса, мы можем принять правильные решения о его реализации. В частности, нам потребуется ввести в класс следующий защищенный объект:

UnboundedOrderedCollection<Assumption*> statements;

Этот объект также позаимствован нами из библиотеки фундаментальных классов главы 9.

Теперь обратимся к классу **Alphabet** (алфавит). Он содержит данные об алфавитах исходного текста и шифра, а также о соответствии между ними. Эта информация необходима для того, чтобы источники знаний могли узнать о выявленных соответствиях между буквами шифра и текста и тех, которые еще предстоит найти. Например, если уже доказано, что буква с а шифре соответствует букве и исходного текста, то это соответствие фиксируется в алфавите и источники знаний уже не будут делать других предположений в отношении буквы м исходного текста. Для эффективности обработки полезно получать данные о соответствии букв шифра и текста двумя способами: по букве шифра и по букве исходного текста. Определим класс **Alphabet** следующим образом:

```
class Alphabet : public BlackboardObject {
public:
      char plaintext (char) const;
      char ciphertext(char) const;
      int isBound(char) const;
};
```

Так же, как и в класс **CipherLetter**, в класс **Alphabet** необходимо включить защищенный объект **affirmation**s и определить операции доступа к его состоянию.

Наконец, определим класс **Blackboard**, который является коллекцией экземпляров класса **Blackboardobject** и его подклассов:

```
class Blackboard : public DynamicCollection<BlackboardObject*>
...
```

Поскольку доска есть разновидность коллекции (тест на наследование), мы предпочитаем образовать этот класс методом наследования, а не с помощью включения экземпляра класса **DynamicCollectlon**. Операции включения в коллекцию и исключения из нее наследуются от класса **Collection**, а следующие пять операций, специфичных для информационной доски, вводятся нами:

- **reset** Очистить доску
- **assertProblem** Поместить на доске начальные условия задачи
- **connect** Подключить к доске источник знании
- **issolved** Истинно, если предложение расшифровано
- **retriaveSolution** Значение расшифрованного текста

Вторая операция устанавливает зависимость между доской и источником знании. На рис. 11-3 приведена итоговая диаграмма классов, связанных с **Blackboard**. Она в первую очередь отражает отношения наследования. Отношения использования (например, между **Assumption** и информационной доской) для простоты опушены.

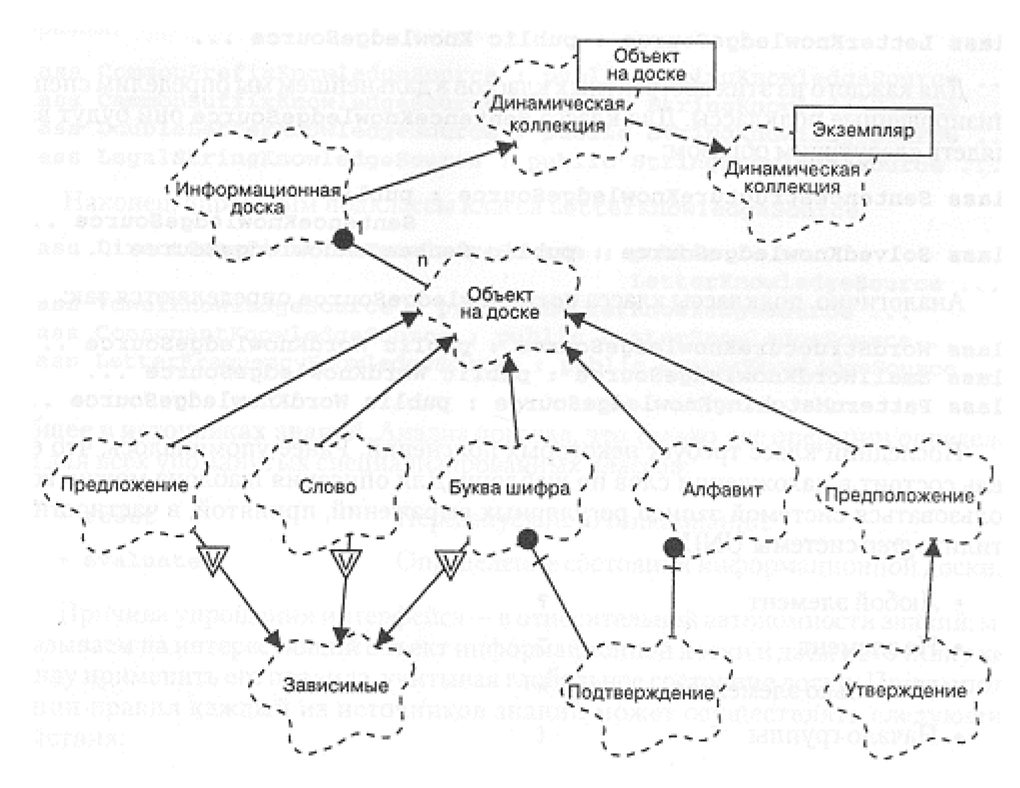


Рис. 11-3. Диаграмма классов информационной доски

Обратите внимание на то, что класс **`Blackboard`** одновременно и инстанциру-ет от шаблона Dynamic**`Collection`**, и наследует от него. Кроме того, становится понятным использование класса **`Dependent`** в качестве примеси. Не привязывая этот класс жестко к иерархии **`Blackboard`**, мы повышаем шансы на его последующее повторное использование.

Проектирование источников знаний

В предыдущем разделе мы выделили тринадцать источников знаний, относящихся к решаемой задаче. Теперь можно приступить к проектированию структур классов для них (как это было сделано для информационной доски) и обобщению их в более абстрактные классы.

Проектирование специализированных источников знаний. Предположим, что существует абстрактный класс **`KnowledgeSource`** (по аналогии с классом **`BlackboardObject`**). Прежде чем определять все тринадцать источников в качестве подклассов одного общего суперкласса, нужно посмотреть, не группируются ли они каким-нибудь образом. Действительно, такие группы находятся: некоторые источники знаний оперируют целым предложением, другие - словами, фрагментами слов или отдельными буквами. Отразим этот факт в следующих определениях:

```
class SentenceKnowledgeSource : public KnowledgeSource ...
class WordKnowledgeSource : public KnowledgeSource ...
class LetterKnowledgeSource : public KnowledgeSource ...
```

Для каждого из этих абстрактных классов в дальнейшем мы определим специализированные подклассы. Для класса **`SentenceKnowledgeSource`** они будут выглядеть следующим образом:

```
class SentenceStructureKnowledgeSource : public
      SentenceKnowledgeSource ...
class SolvedKnowledgeSource : public
      SentenceKnowledgeSource ...
```

Аналогично, подклассы класса **WordKnowledgeSource** определяются так:

```
class wordStructureKnowledgeSource : public
WordKnowledgeSource ...
class SmallWordKnowledgeSource : public WordKnowledgeSource
...
class pattemMatchingKnowledgeSource : public
WordKnowledgeSource ...
```

Последний класс требует некоторых пояснений. Ранее упоминалось, что его цель состоит в нахождении слов по шаблону. Для описания шаблона можно воспользоваться системой записи регулярных выражении, принятой, в частности, в утилите *grep* системы UNIX:

- Любой элемент ?
- Не элемент ~
- Несколько элементов *
- Начало группы {
- Конец группы }

Используя такие обозначения, мы можем передать объекту этого класса шаблон **?E~{A E I O U}**, чтобы он искал в своем словаре слово из трех букв, начинающееся с некоторой буквы, после которой идет **E**, а затем - любая буква кроме гласной.

Поскольку проверка по шаблону является методом, полезным как для данной системы в целом, так и в других областях, соответствующий класс целесообразно выделить в качестве самостоятельной абстракции. Поэтому неудивительно, что мы воспользуемся классом из нашей библиотеки (см. главу 9). В результате наш класс для проверки по шаблону будет выглядеть следующим образом:

```
class PatternMatchingKnowledgeSource : public
WordKnowledgeSource { public:
      ...
protected:
      static BoundedCollection<Word*> words;
      REPattemMatching pattemMatcher;
};
```

Все экземпляры этого класса разделяют общий словарь, но каждый из них может иметь собственного агента для сравнения с шаблонами.

На данном этапе проектирования подробности реализации этого класса для нас не существенны, поэтому мы не будем на них подробно останавливаться.

Определим теперь подклассы класса **stringKnowledgeSource** следующим образом:

```
class CoinmonPrefixKnowledgeSource : public
StringKnowledgesource ...
class ConunonSuffixKnowledgeSource : public
StringKnowledgesource ...
class DoubleLetterKnowledgeSource : public
StringKnowledgesource ...
class LegalStringKnowledgeSource : public
StringKnowledgesource ...
```

Наконец, определим подклассы класса **LetterKnowledgeSource**:

```
class DirectSubstitutionKnowledgeSource : public
      LetterKnowledgeSource ...
class VowelKnowledgeSource : public
      LetterKnowledgeSource ...
class ConsonantKnowledgeSource : public
      LetterKnowledgeSource ...
```

```
class LetterPrequencyKnowledgeSource : public
      LetterKnowledgeSource ...
```

Общее в источниках знаний. Анализ показал, что только две операции определены для всех упомянутых специализированных классов:

- **Reset** Перезапуск источника знаний.
- **evaluate** Определение состояния информационной доски.

Причина упрощения интерфейса - в относительной автономности знаний: мы указываем на интересующий объект информационной доски и даем источнику команду применить его правила, учитывая глобальное состояние доски. При выполнении правил каждый из источников знаний может осуществлять следующие действия:

- Высказать предположение о подстановке.
- Найти противоречие в ранее предложенных подстановках и откатить их.
- Высказать утверждение о подстановке.
- Сообщить контроллеру о своем желании записать на доску что-то интересное.

Все эти действия являются общими для всех источников знаний. Перечисленные операции образуют механизм вывода заключений. Определим механизм вывода (**InferenceEngine**) как объект, который выполняет известные правила для того, чтобы либо найти новые правила (прямая последовательность рассуждений), либо Доказать некоторую гипотезу (обратная последовательность рассуждений). На основании сказанного введем следующий класс:

```
class InferenceEngine ( public:
      InferenceEngine<DynamicSet<Rules*>};
      ...
};
```

Конструктор класса создает экземпляр объекта и населяет его правилами. Лишь одна операция сделана в этом классе видимой для источников знании:

- **evaluate** Выполнить правило механизма вывода.

Теперь о том, как сотрудничают источники знаний: каждый специализированный источник определяет свои собственные правила и возлагает ответственность за их выполнение на класс **InferenceEngine**. Точнее, операция **KnowledgeSource::evaluate** вызывает метод **InferenceEngine::evaluate**, что приводит к выполнению одной из четырех упомянутых выше операций. На рис. 11-4 показан сценарий такого взаимодействия:

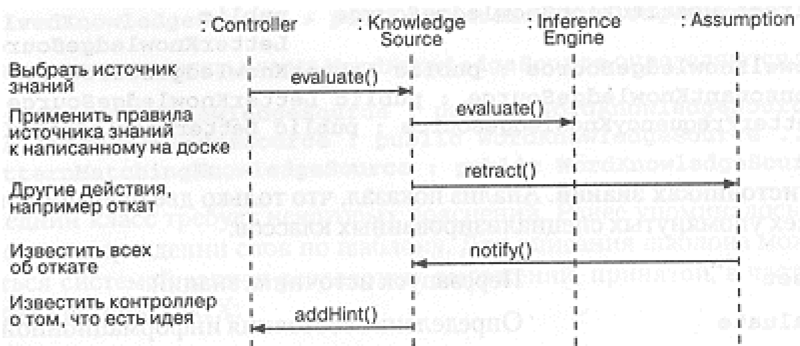


Рис. 11-4. Взаимодействия с источником знаний

Что такое правило? Для иллюстрации приведем (в формате Lisp) правило, касающееся знаний об общеупотребительных суффиксах:

```
((* I ? ?)
(* I N G)
(* I E S)
```

```
(* I E D))
```

Это правило означает, что заданному шаблону **`*I??`** (условие - **`antecedent`**) могут соответствовать суффиксы **`ING`**, **`IES`** и **`IED`** (заключение - **`consequent`**). В С++ можно определить следующий класс для представления правил:

```
class Rule{ public:
      ...
      int bind(String<char>& antecedent, String<char>&
consequent);
      int   remove(Strlng<char>& antecedent);
      int   remove(String<char>t antecedent, String<char>&
conseiruent);
      int   hasConflict(const String<char>& antecedent)
      const;
protected:
      String<char> antecedent;
      List<String<char> > consequents;
};
```

Смысл приведенных операций полностью понятен из их наименований. Мы здесь повторно использовали некоторые классы из главы 9.

С точки зрения строения данного класса можно утверждать, что источники знаний являются разновидностью механизма вывода. Кроме того, они ассоциированы с объектами доски, поскольку находят там приложение своим усилиям. Наконец, каждый источник знаний связан с контроллером и посылает ему свои соображения. Контроллер, в свою очередь, может активизировать источники знаний.

Выразим все сказанное следующим образом:

```
class KnowledgeSource :   public InferenceEngine,
      public Dependent (
public:
      KnowledgeSource(Blackboard*, Controller*);
      void reset() ;
      void evaluate {} ;
protected:
      Blackboard* blackboard;
      Controller* controller;
      UnboundedOrderedCollection<Assumption*> pastAssumptions;
};
```

В этот класс введен защищенный элемент данных **`pastAssumptions`**, позволяющий сохранять всю историю предположений в целях самообучения.

Экземпляры класса **`Blackboard`** служат для хранения объектов информационной доски. По схожим соображениям, необходим также класс **`KnowledgeSources`**, охватывающий все источники знаний, относящиеся к решаемой задаче:

```
      class KnowledgeSources : public
DynamicCollection<KnowledgeSource*>...
```

Одно из свойств этого класса состоит в том, что при создании его экземпляра создаются также 13 специализированных источников знаний. Для объектов этого класса определяются три операции:

- **`restart`** Перезапустить источник знаний.
- **`StartKnowledgeSource`** Задать начальные условия для источника знаний.
- **`connect`** Связать источник знаний с доской или контроллером.

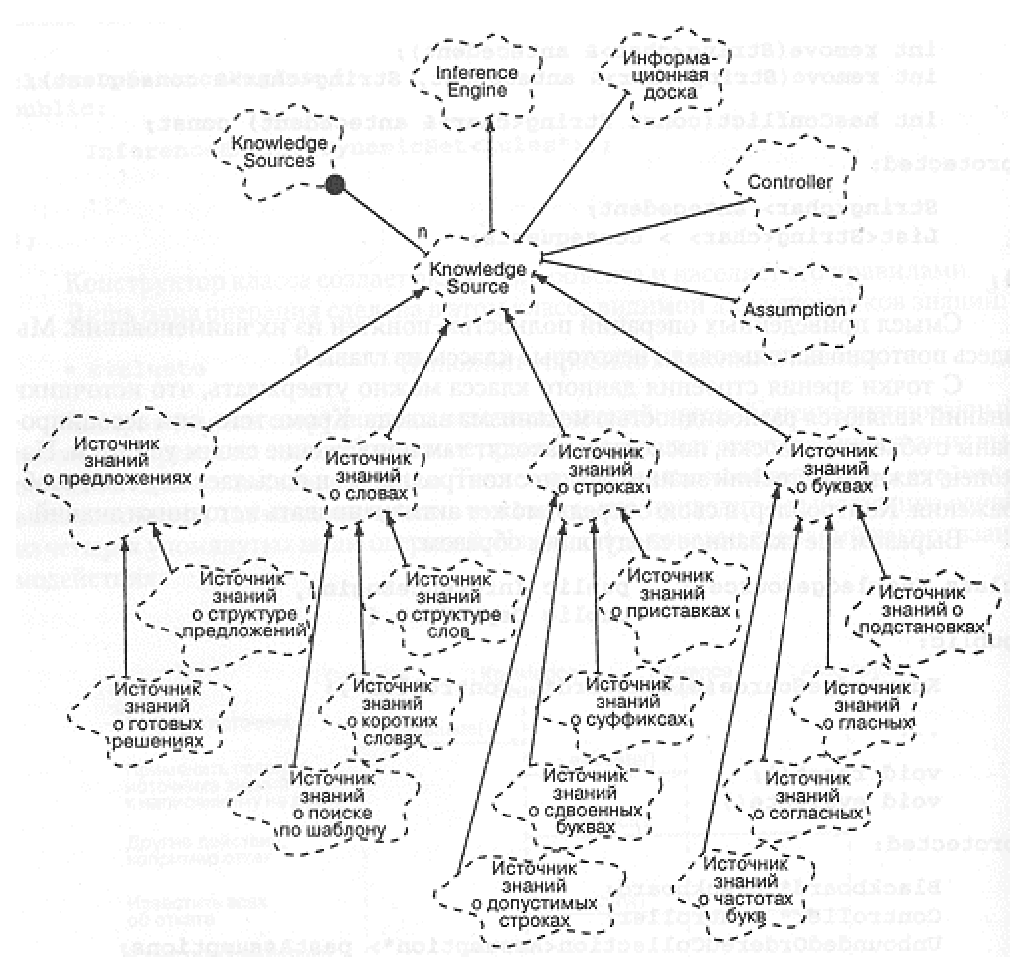


Рис. 11-5. Диаграмма классов источников знаний

На рис. 11-5 показана структура созданных в процессе проектирования классов источников знаний.

Проектирование контроллера

Рассмотрим более подробно взаимодействие контроллера с отдельными источниками знаний. В процессе поэтапной расшифровки криптограммы отдельные источники знаний выявляют полезную информацию и сообщают ее контроллеру. С другой стороны, может быть обнаружено, что ранее переданная информация оказалась ложной и ее надо устранить. Поскольку все источники знаний имеют равные права, контроллер должен опросить их все, выбрать тот, информация которого кажется наиболее полезной, и разрешить ему внести изменения вызовом его операции **`evaluate`**.

Каким образом контроллер определяет, какой из источников знаний следует активизировать? Можно предложить несколько разумных правил:

- Утверждение более приоритетно чем предположение.
- Если кто-то говорит, что решил всю фразу, надо дать ему возможность высказаться.
- Проверка по шаблону более приоритетна, чем источник, анализирующий структуру предложения.

Контроллер действует в качестве агента, ответственного за взаимодействие источников знаний.

Контроллер должен быть в ассоциативной связи с источниками знаний через класс **`KnowledgeSources`**. Кроме того, он должен иметь в качестве одного из своих свойств коллекцию высказываний, упорядоченных по

приоритету. Тем самым контроллер легко может выбрать для активизации источник знаний с наиболее интересным высказыванием.

После изолированного анализа класса мы предлагаем ввести для класса controller следующие операции:

- **reset** Перезапуск контроллера.
- **addHint** Добавить высказывание от источника знаний.
- **removeHint** Удалить высказывание от источника знаний.
- **processNextHint** Разрешить выполнение следующего по приоритету высказывания.
- **isSolved** Селектор. Истина, если задача решена.
- **UnableToProceed** Селектор. Истина, если источники знаний застряли.
- **connect** Устанавливает связь с источником знаний.

Все эти решения можно описать следующим образом:

```
class Controller {
public:
    ...
    void reset{);
    void connect(Knowledgesource&);
    void addHint{KnowledgeSourceu);
    void removeHint(KnowledgeSourceu) ;
    void processNextHint () ;
    int isSolved() const;
    int unableToProceed() const;
};
```

Контроллер в некотором смысле управляется источниками знаний, поэтому Для описания его поведения наилучшим образом подходит схема конечного автомата.

Рассмотрим диаграмму состояний и переходов на рис. 11 -6. Из нее видно, что контроллер может находиться в одном из пяти основных состояний: инициализация (**Initializing**), выбор (**Selecting**), вычисление (**Evaluating**), тупик (**Stuck**) и решение (**Solved**). Наибольший интерес для нас представляет поведение контроллера при переходе от выбора к вычислению. В состоянии **selecting** контроллер переходит от создания стратегии (**CreatingStrategy**) к вычислению высказывания (**ProcessingHint**) и, в конце концов, выбирает источник знаний (**SelectingKS**).

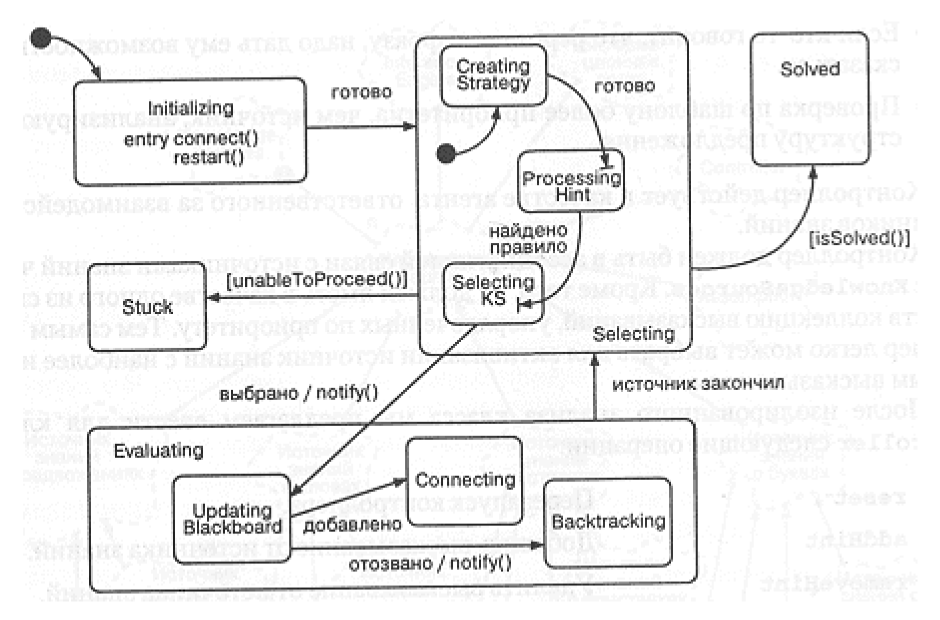


Рис. 11-6. Контроллер как конечный автомат

Дав одному из источников возможность высказаться, контроллер переходит в состояние **Evaluating**, где прежде всего изменяет состояние информационной доски. Это вызывает переход в состояние **Connect**ing при добавлении источника знании или к Baclctraking, если предположение не оправдалось и надо откатить его, оповестив при этом все зависимые источники знаний.

Конечной точкой работы нашего механизма является **solved** (задача решена) или **stuck** (тупиковая ситуация).

11.3. Эволюция

Интеграция

Теперь, когда ключевые абстракции предметной области выявлены, можно приступить к их соединению в действующее приложение. Мы будем реализовывать и проверять вертикальные срезы системы, а затем последовательно отрабатывать механизмы.

Интеграция объектов верхнего уровня. На рис. 11-7 показана диаграмма объектов нашей системы на самом верхнем уровне, которая полностью соответствует структуре информационной доски, приведенной на рис. 11-1. Физическое содержание объектов доски в коллекции **theBlackboard** и источников знаний в коллекции the**KnowledgeSources** показано в соответствии с описанием вложенности классов.

На диаграмме появился экземпляр класса **Cryptographer**. Он агрегирует объекты доски, источники знаний и контроллер. В результате наша программа может

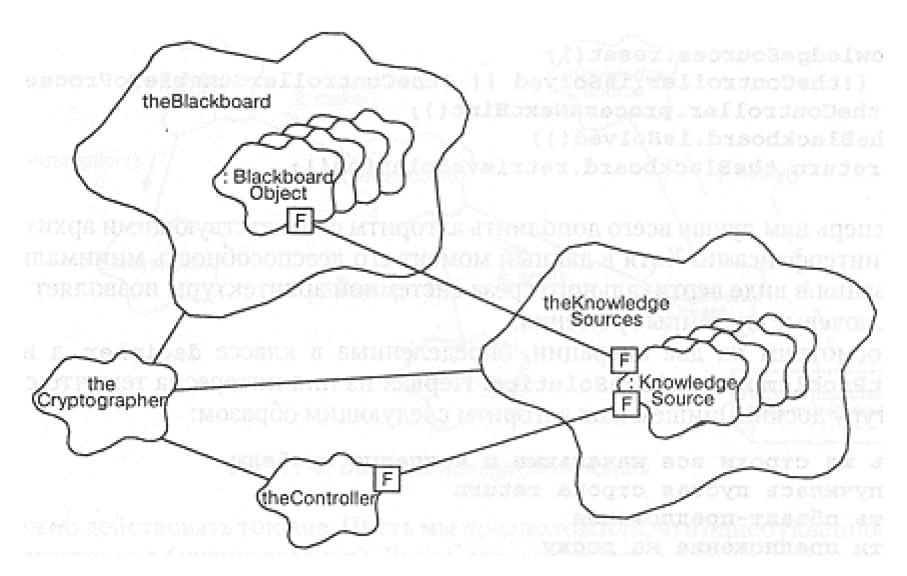


Рис. 11-7.Диаграмма объектов криптоанализа

иметь несколько экземпляров этого класса, а следовательно и несколько досок, функционирующих параллельно.

Для этого класса следует определить две основные операции:

- **reset** Перезапустить информационную доску
- **decipher** Начать дешифровку криптограммы

Конструктор этого класса должен создать зависимости между доской и источниками знаний, а также между источниками знаний и контроллером. Метод reset предельно прост: его цель состоит в том, чтобы вернуть эти связи и объекты в начальное состояние.

Метод **decipher** принимает строку - криптограмму. Теперь функции высокого уровня нашего приложения становятся предельно простыми, как это обычно и происходит в объектно-ориентированных системах:

```
char* solveProblem (char* ciphertext) {
      Cryptographer theCryptographer;
      return theCryptographer.decipher(ciphertext);
}
```

Метод **decipher** оказывается несколько сложнее. В первую очередь с помощью операции **assertProblem** задание помещается на доску. После этого активизируются источники знаний. И, наконец, начинается циклический процесс обращения источников знаний к контроллеру с новыми и новыми предположениями и утверждениями до тех пор, пока не будет найдено решение задачи либо процесс не зайдет в тупик. Для иллюстрации можно воспользоваться диаграммами взаимодействия или диаграммами объектов, но код на C++ выглядит тоже не слишком сложно:

```
theBlackboard.assertProblem();
cheKnowledgeSources.reset() ;
while (StheController.isSolvea ||
!theController.unableToProceed() )
      theController.proceaaNextHint();
it (theBlackboard.isSolved())
      return theBlackboard.retrieveSolution() ;
```

Теперь нам лучше всего дополнить алгоритм соответствующими архитектурными интерфейсами. Хотя в данный момент его дееспособность минимальна, но реализация в виде вертикального среза системной архитектуры позволяет проверить ключевые системные решения.

Посмотрим на две операции, определенные в классе **decipher**, а именно **assertProblem** и **retrieveSolution**. Первая из них интересна тем, что создает структуру доски. Опишем наш алгоритм следующим образом:

```
        убрать из строки все начальные и концевые пробелы if
получилась пустая строка return создать объект-предложение
занести предложение на доску создать объект-слово (самое
крайнее слева) занести слово на доску добавить слово к
предложению for каждый символ строки слева направо if символ
есть пробел
        сделать текущее слово предыдущим
        создать объект-слово
        занести слово на доску
        добавить слово к предложению else
        создать объект "буква шифра"
        занести букву на доску
        добавить букву к слову
```

В главе 6 уже упоминалось, что целью проектирования является создание наброска реализации. Эта запись представляет достаточно детализированный алгоритм, так что показывать его полную реализацию на С++ нет необходимости.

Операция **`retrieveSolution`** очень проста: она возвращает строку,записанную в данный момент на доске. Вызывая эту операцию до того как функция **`isSolved`** вернула значение **`True`**, можно получать частичные решения.

Реализация механизма предположений. Итак, мы умеем устанавливать и извлекать значения объектов доски. Теперь нам нужен механизм выдвижения высказывании об этих объектах. Этот механизм интересен ввиду его динамичности. При поиске решения предположения непрерывно создаются и отзываются, чем как раз и приводится в действие весь процесс.

На рис. 11-8 показан сценарий выдвижения предположений. Источник знаний сообщает об имеющихся предположениях информационной доске, которая применяет их к алфавиту и оповещает остальные источники.

В простейшем случае, чтобы отменить предположение, мы просто прокручиваем этот механизм в другую сторону. Например, чтобы отменить предположение о букве, мы убираем из ее коллекции все предположения вплоть до неверного.

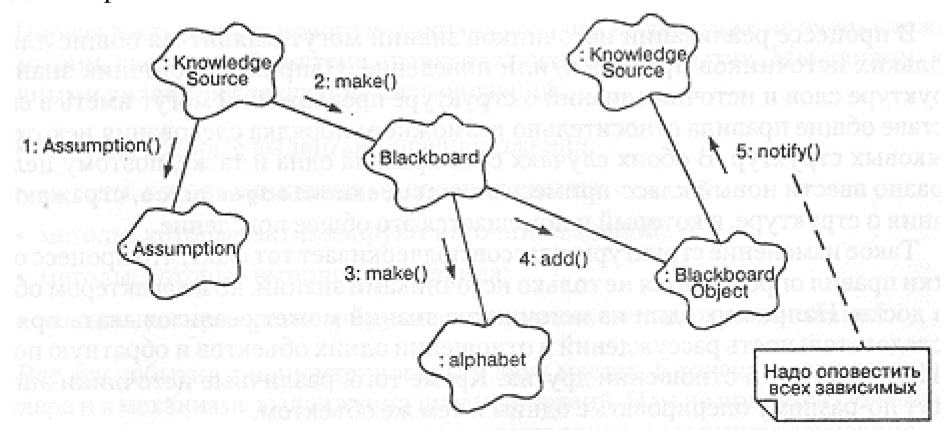


Рис. 11-8. Выдвижение предположений

Можно действовать тоньше. Пусть мы предположили, что однобуквеиное слово соответствует **`I`** (нужна гласная). Далее, сделано предположение, что некоторое двухбуквенное сочетание - это **`NN`** (нужны согласные). Если первое предположение окажется ошибочным, то второе вполне может быть сохранено. При таком подходе класс **`Assumption`** нужно дополнить еще одним методом, регистрирующим связь предположений между собой (взаимозависимость). Реализацию этого поведения можно отложить на более поздний срок, поскольку оно мало влияет на архитектуру.

Добавление источников знаний

Теперь, когда определены ключевые абстракции информационной доски и механизмы выдвижения и проверки предположений, необходимо реализовать механизм вывода (класс **InferenceEngine**), связывающий все источники знаний в единое целое. Ранее уже упоминалось, что механизм вывода должен реализовать одну основную операцию, а именно выполнение правила, **evaluateRules**. Мы не будем на этом подробно останавливаться, поскольку реализация не влияет на проектные решения.

Убедившись в правильной работе механизма вывода, можно последовательно вводить в систему источники знаний. Целесообразность именно такого процесса объясняется двумя причинами:

- Трудно заранее выяснить, какие правила существенны для каждого из источников знаний, не испытав систему на конкретной задаче.
- Отладка базы знаний существенно упрощается при последовательном добавлении правил.

Реализация источников знаний является предметом инженерии знаний. Для построения конкретного источника знаний требуется консультация с экспертами (например, криптографами). При анализе источников знаний может выявиться, что одни правила бесполезны, другие слишком специализированы или излишне обобщены, а некоторых явно недостает. После анализа правила источника могут модифицироваться. Иногда требуется создание нового источника знаний.

В процессе реализации источников знании могут выявиться общие для нескольких источников правила и/или поведение. Например, источник знаний о структуре слов и источник знаний о структуре предложений могут иметь в своем составе общие правила относительно возможного порядка следования некоторых языковых структур. В обоих случаях суть правила одна и та же, поэтому целесообразно ввести новый класс-примесь **StructureKnowledgeSource**, отражающий знания о структуре, в который и помещается это общее поведение.

Такое изменение структуры классов подчеркивает тот факт. что процесс обработки правил определяется не только источниками знаний, но и характером объектов доски. Например, один из источников знаний может реализовывать прямую последовательность рассуждений в отношении одних объектов и обратную последовательность - в отношении других. Кроме того, различные источники знаний могут по-разному оперировать с одним и тем же объектом.

11.4. Сопровождение

Расширение функциональных возможностей

В этом разделе мы попытаемся улучшить возможности проектируемой системы и оценить ее гибкость.

В интеллектуальных системах очень важно наряду с решением задачи получить информацию о самом процессе поиска решения. Для этого нужно придать системе способность самоанализа: регистрировать ход активизации источников знаний, причины и характер выдвигаемых предположений и т. д., чтобы иметь возможность запросить у системы, по какой причине сделано конкретное предположение и к каким результатам оно приводит.

Для реализации такого свойства необходимо сделать две вещи. Во-первых, нужно ввести механизм трассировки действий контроллера и источников знаний, а во-вторых - модифицировать некоторые методы, чтобы они записывали соответствующую информацию. Идея состоит том, что действия источников знаний и контроллера регистрируются в некотором общем центральном хранилище.

Посмотрим, какие классы нам понадобятся. Прежде всего, введем класс Action, регистрирующий действия источников знаний и контроллера:

```
class Action ( public:
      Action(KnowledgeSource* who, BlackboardObject* what,
char* why);
      Action<Controller* who, KnowledgeSource* what, char*
why);
};
```

Экземпляр данного класса создается, например, при активизации контроллером какого-либо источника знаний. При этом в аргумент **`who`** (кто) заносится указатель на контроллер, в аргумент **`what`** (что) - активный источник знаний, а в аргумент **`why`** (почему) - какое-либо пояснение (например, приоритет предположения).

Первая часть нашего нового механизма создана, вторая тоже не очень сложна. Посмотрим, где в нашей системе происходят основные события. Мы увидим, что основными являются следующие пять операций:

- методы, которые выдвигают предположения;
- методы, которые откатывают предположения;
- методы, которые активизируют источники знаний;
- методы, которые выполняют правила;
- методы, которые регистрируют высказывания от источников знаний.

Все эти события сконцентрированы в двух местах: в конечном автомате контроллера и в механизме выдвижения предположений. Нам не придется существенно изменять архитектуру системы, чтобы учесть указанные выше требования.

Для полноты нам остается только создать объект, отвечающий на вопросы пользователя системы: кто? что? когда? почему?. Спроектировать такой объект несложно, поскольку вся нужная для его работы информация может быть получена от экземпляров класса **`Actions`**.

Изменение технических требований

Если принятые проектные решения были реализованы правильно, то новые технические требования к системе могут быть удовлетворены при минимальных изменениях проекта. Допустим, что предъявлены три новые требования к данной системе:

- возможность дешифровки с иностранных языков;
- возможность дешифровки перестановочного и простого подстановочного шифра, использующего (одну) подстановку и перестановку;
- способность к самообучению.

Первое требование самое простое, поскольку связь нашей системы с английским языком не является существенной. Она отражается только на правилах источников знаний. Даже класс **`Alphabet`** сделан независимым от конкретного национального алфавита.

Второе требование существенно сложнее, но разрешимо в рамках механизма доски. Это потребует введения новых источников знаний относительно шифров перестановки. Ключевые механизмы и абстракции при этом также полностью сохранятся, но потребуется введение новых классов, которые будут действовать в рамках существующих механизмов выдвижения предположений и вывода.

Труднее всего выполнить последнее требование, так как обучение компьютеров относится к области искусственного интеллекта. Можно, например, предложить контроллеру в тупиковых ситуациях обращаться за помощью к пользователю системы с просьбой выдвинуть предположение. Такие предположения (вкупе с последовательностью действий, которая завела

в тупик) могут регистрироваться системой и позволят в дальнейшем избегать подобных тупиков. Такой простейший механизм обучения может быть введен в нашу систему без существенного изменения структуры классов и в рамках действующих механизмов.

Дополнительная литература

При рассмотрении архитектурных шаблонов Шоу (Shaw) [A 1991] обсуждает метафору информационной доски и другие базовые идеи.

Енглемор и Морган (Englemore and Morgan) [C 1988] дали исчерпывающее обсуждение информационных досок, включая их эволюцию, теорию, проектирование и приложение. Существует описание двух объектно-ориентированных систем информационных досок: BB1 из Стэнфорда и BLOB, разработанной для Британского министерства обороны. Другие полезные сведения относительно информационных досок могут быть найдены у Хайеса-Рота (Hayes-Roth) D 1985] и Нии (Nil) [) 1986].

Подробное обсуждение индуктивного и дедуктивного подходов в системах формального вывода можно найти в работах Барра и Фейгенбаума (Barr and Feigenbamn) [J 1981], Брахмана и Левескье (Brachman and Levesque) U 1985], Хайес-Рота, Ватермана и Лена (Hayes-Roth, Waterman, and Lenat) [J 1983], а также Винстона и Хорна (Winston and Horn) [Gl 989].

Мейер и Матиас (Meyer and Matyas) [I 1982] рассмотрели сильные и слабые стороны разных шифров и алгоритмы их дешифровки.

Глава 12

Управление: контроль за движением поездов

Программная индустрия развилась настолько, что охватывает многие новые области приложений: от встроенных микрокомпьютеров для управления двигателем автомобиля до выполнения рутинной работы при изготовлении фильмов и обеспечения интерактивного доступа миллионов телезрителей к базам видеоинформации. Отличительной особенностью таких больших систем является их чрезвычайная сложность. Конечно, построить компактную реализацию системы - задача почетная, но некоторые большие задачи несомненно требуют большого объема кода. В крупных проектах нередко участвуют программистские организации в сотни человек, которые должны написать миллионы строк кода. Программы должны удовлетворять требованиям, неизбежно меняющимся в процессе работы. Как правило, в рамках таких проектов создается не одна программа, работающая на одном компьютере, а комплекс программ, функционирующих в параллельной распределенной среде на нескольких компьютерах, связанных между собой разнообразными каналами передачи информации. Для того, чтобы уменьшить вероятность неудачи, в таких проектах предусматривается обычно центральная организация, отвечающая за архитектуру и целостность системы. Некоторые части системы нередко выполняются по субконтрактам другими компаниями. Таким образом, команда разработчиков никогда не собирается вместе, она распределена в пространстве и, - так как в больших проектах происходит постоянное обновление кадров, - во времени.
Если за создание большой системы возьмется разработчик, который занимался написанием в оконной среде небольших программ, рассчитанных на одного пользователя, его несомненно испугают возникающие проблемы; возможно, даже настолько, что он сочтет глупостью попытку создать такую программу. Но действительность такова, что большие системы должны строиться. И в некоторых случаях глупо не попытаться. Вообразим себе ручное управление авиационными полетами во круг столичного аэропорта, систему жизнеобеспечения космической станции, зависящую от "человеческого фактора" или ведение учета в международном банке, выполняемое на счетах. Успешная автоматизация таких систем приводит не только к решению очевидных проблем, но и приносит множество неожиданных выгод: снижение эксплуатационных расходов, повышение надежности, увеличение функциональных возможностей. Конечно же, ключевое слово здесь - успешная. Из всего сказанного понятно, что создание больших систем - чрезвычайно трудная задача. Поэтому при ее решении необходимо применять все лучшее из инженерной практики и использовать интуицию ведущих проектировщиков.
В этой главе представлена как раз такая задача. Она демонстрирует, как объектно-ориентированное проектирование облегчает выполнение сверхбольших программных проектов.

12.1. Анализ

Определение границ проблемной области

Для большинства люден, живущих в США, поезда являются символом давно ушедшей эпохи. В Европе и странах Востока ситуация совершенно противоположная. В отличие от США, в Европе мало национальных и международных автомобильных магистралей, а цены на бензин и газ сравнительно высоки. Поэтому поезда составляют основу транспортной сети континента; по десяткам тысяч километров путей ежедневно перевозится множество людей и грузов - и в отдельных городах, и между различными странами. Ради справедливости отметим, что в США поезда играют по-прежнему важную роль в перевозке грузов. С разрастанием городов их центры становятся все более и более перегруженными, и на легкий рельсовый

транспорт возлагаются надежды решить проблему перегрузки и загрязнения окружающей среды двигателями внутреннего сгорания.

Железные дороги по-прежнему являются коммерческими и, следовательно, они должны быть прибыльными. Железнодорожные компании обязаны постоянно поддерживать баланс между требованиями экономии и безопасности и нарастающей интенсивностью перевозок с одной стороны и эффективным и предсказуемым расписанием - с другой. Эти противоречия наводят на мысль, что решения об управлении движением поездов необходимо принимать автоматически, и, в том числе, производить контроль за всеми элементами железной дороги с помощью компьютера.

Такие автоматические и полуавтоматические системы сегодня существуют в Швеции, Великобритании, Германии, Франции и Японии [I]. Подобная система, называемая Продвинутой Системой Управления Железнодорожным Транспортом, была разработана в Канаде и США с участием следующих компаний: Amtrak, Burlington, Canadian National Railway Company, CP Rail, CSX Transportation, Norfolk and Western Railway Company, Southern Railway Company, Union Pacific. Эффект от каждой из этих систем был и экономический, и социальный; результа-

Требования к системе управления движением

Система управления движением выполняет две главные функции: выбор маршрутов железнодорожных перевозок и контроль систем, обеспечивающих перевозки. Эти функции включают: планирование перевозок, контроль местонахождения поездов, контроль за перевозками, предотвращение конфликтов, прогнозирование нарушении, регистрацию всех операций. На рис. 12-1 показана схема основных элементов системы управления движением [2].

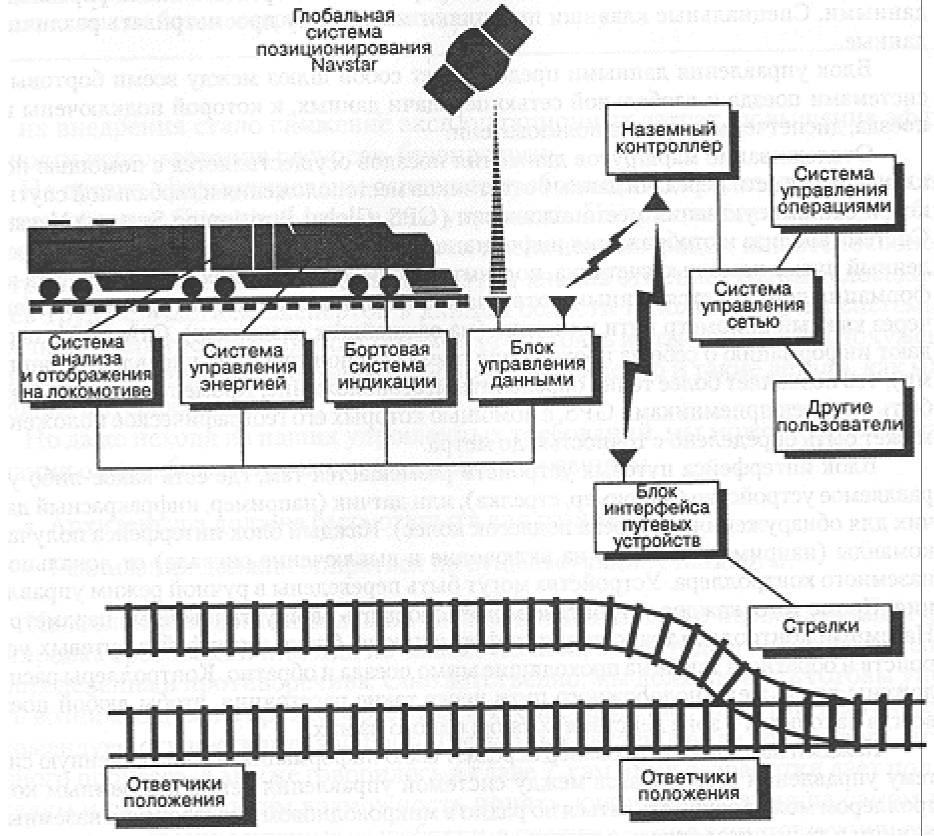


Рис. 12-1. Система управления движением

Система анализа и отображения информации на локомотиве состоит из множества дискретных и аналоговых датчиков для контроля за такими параметрами, как температура, давление масла, количество топлива, напряжение и сила тока на генераторе, число оборотов вала двигателя в

минуту, температура воды, тяговая мощность. Значения параметров с датчиков поступают к машинисту через дисплейную систему, а к диспетчеру и обслуживающему персоналу вне поезда - через сеть. Предупреждение или сигнал тревоги выдается и регистрируется всякий раз, когда показания датчика выходят за пределы нормального режима. Журнал показаний датчиков используется при проведении эксплуатационных работ и для управления расходом топлива.

Система управления энергией в режиме реального времени подсказывает инженеру поезда, как наиболее эффективно использовать установку. Входными данными для этой системы являются: профиль и качество пути, ограничения по скорости, расписание, загрузка поезда, максимальная развиваемая мощность. Исходя из этих данных, система может определить оптимальный по расходу топлива режим работы двигательных установок, согласующийся с заданным расписанием и требованиями безопасности. Рекомендации системы, профиль и качество пути, местоположение и скорость поезда могут отображаться с помощью бортовой системы индикации.

Бортовая система индикации обеспечивает человеко-машинный интерфейс для машиниста. На нее может выводиться информация из системы анализа и отображения информации на локомотиве, системы управления энергией и блока управления данными. Специальные клавиши позволяют машинисту просматривать различные данные.

Блок управления данными представляет собой шлюз между всеми бортовыми системами поезда и глобальной сетью передачи данных, к которой подключены все поезда, диспетчеры и прочие пользователи.

Отслеживание маршрутов движения поездов осуществляется с помощью подключенных к сети передачи данных ответчиков местоположения и глобальной спутниковой системы указания местоположения (GPS, Global Positioning System) Navstar. Система анализа и отображения информации на локомотиве может вычислять пройденный путь с помощью счетчика, подсчитывающего число оборотов колеса. Эта информация дополняется данными ответчиков местоположения, которые размещены через каждый километр пути или чаще (на важнейших развилках). Ответчики передают информацию о себе на проходящие поезда (используя блок управления данными), что позволяет более точно определить местоположение. Кроме того, поезд может быть оснащен приемниками GPS, с помощью которых его географическое положение может быть определено с точностью до метра.

Блок интерфейса путевых устройств размещается там, где есть какое-либо управляемое устройство (например, стрелка), или датчик (например, инфракрасный датчик для обнаружения перегрева подвесок колес). Каждый блок интерфейса получает команды (например, команды на включение и выключение сигнала) от локального наземного контроллера. Устройства могут быть переведены в ручной режим управления. Кроме того, каждое устройство может сообщать свои установочные параметры. Наземный контроллер транслирует информацию на блоки интерфейса путевых устройств и обратно, а также на проходящие мимо поезда и обратно. Контроллеры расположены вдоль железнодорожного пути через такие расстояния, чтобы любой поезд всегда находился в зоне действия хотя бы одного из них.

Каждый наземный контроллер передает свою информацию на объединенную систему управления сетью. Связь между системой управления сетью и наземным контроллером может осуществляться по радио в микроволновом диапазоне, по наземным линиям или по оптоволокну в зависимости от удаленности данного контроллера. Система управления сетью обеспечивает функционирование всей сети. Она может автоматически направлять информацию по другому маршруту в сети, если на одном из путей произойдет отказ оборудования,

Система управления сетью, в свою очередь, подсоединяется к одному или нескольким диспетчерским центрам, которые объединены в систему управления операциями. Система управления сетью соединена и с другими пользователями. В системе управления операциями диспетчеры могут задавать маршруты поездов и отслеживать их передвижение. Для управления различными участками выделяются отдельные диспетчеры; каждая диспетчерская управляющая консоль отвечает за одну или несколько территорий. Маршрутизация поездов подразумевает выдачу инструкций для автоматического перевода поезда с пути на путь, установку ограничения скорости, управление пропуском автомобилей на переездах, разрешение и запрещение движения поезда в зависимости от занятости определенных участков пути. Диспетчеры могут наблюдать за состоянием путей впереди по маршруту поезда и передовать эту информацию машинисту. Поезда могут быть остановлены системой управления операциями (вручную диспетчерами или автоматически), когда обнаруживается опасность (выход поезда из графика, повреждение пути, возможность столкновения). Диспетчеры могут также вызвать на экран любую информацию, доступную машинистам отдельных поездов, разослать распоряжения по движению, установить параметры путевых устройств и пересмотреть план движения.

Расположение путей и путевое оборудование могут со временем меняться. Число поездов и маршруты их движения могут изменяться ежедневно. Система должна обеспечивать возможность подключения новых датчиков, сетей и оборудования, выполненных по более совершенным технологиям.

том их внедрения стало снижение эксплуатационных затрат, повышение эффективности использования ресурсов, безопасность.

На врезке сформулированы основные требования к системе управления движением поездов. Очевидно, они сильно упрощены. На практике детальные требования к большой системе вырабатываются после демонстрации жизнеспособности программного решения проблемы. При этом анализ отменяет сотни человеко-месяцев труда с участием экспертов в данной области и пользователей системы. В конечном счете требования к системе могут состоять из тысяч страниц документации, специфицирующей не только базовое поведение, но и такие детали, как макеты форм интерфейса.

Но даже исходя из наших упрощенных требований, мы можем сделать два замечания о разработке системы управления движением:

- Архитектура должна быть открыта для развития.
- Реализация должна опираться на существующие стандарты.

Наш опыт разработки больших систем показывает, что первоначальная формулировка требований никогда не бывает полной, она всегда в некоторой степени неопределенна и противоречива. Соответственно, мы должны быть готовы управлять возникающими в процессе разработки неопределенностями. Мы настоятельно рекомендуем осуществлять эволюцию подобных систем в виде пошагового, итеративного процесса. Как уже говорилось в главе 7, сам цикл разработки дает пользователям и разработчикам возможность понять, какие требования на самом деле существенны; именно процесс разработки, а не упражнения в чистописании спецификаций в отсутствии готовой частичной реализации или прототипа. Кроме того, необходимо учитывать, что на создание большой системы может быть затрачено несколько лет. За это время сильно изменится аппаратная часть.[40] Поэтому требования к программе должны предусматривать адаптацию к новой технике. Бессмысленно создавать элегантную архитектуру для аппаратуры, которая гарантированно устареет за время разработки. Мы считаем, что в архитектуру программной системы следует включать только те аппаратные особенности, которые непосредственно опираются на существующие стандарты: связь, сети передачи данных, графику и протокол работы датчиков. Для совершенно

новых систем иногда Приходится становиться первопроходцами аппаратных и программных средств. Это приводит к повышению риска, который для большинства систем и без того высок. Разработка программного обеспечения, особенно, когда речь идет об успешном завершении большого приложения, неизбежно связана с риском, и наша цель - снизить этот риск до минимума.

Очевидно, что мы не сможем подробно рассмотреть все вопросы анализа и проектирования описанной системы в одной главе или даже в одной книге. Так как наша задача - показать, как работают обозначения и методология, сосредоточимся на построении гибкой архитектуры изучаемой области.

Системные и программные требования: хрупкий компромисс

Крупные проекты, подобные рассматриваемому, обычно организуются вокруг небольшой центральной группы, ответственной за глобальную архитектуру системы, а сама разработка передается сторонним субподрядчикам или другим группам внутри той же организации. Уже на стадии анализа системные архитекторы имеют некоторую концептуальную модель, которая разделяет аппаратную и программную части реализации. Многие, правда, считают, что это уже не анализ, а проектирование. Это - спорный вопрос. В самом деле, трудно решить, что показано на схеме рис. 12-1- исходные требования или проект системы. Но в любом случае схема предполагает, что на данной стадии разработки архитектура системы принципиально объектно-ориентированна. Например, на схеме присутствуют такие сложные объекты, как система управления энергией или система управления операциями. Каждый из них выполняет одну из основных функций всей системы. Это как раз то, о чем говорилось в главе 4: объекты самого высокого уровня абстракции отвечают за основные функции системы. Поэтому процесс анализа в данном случае мало отличается от процесса проектирования.

Когда мы уже имеем скелет архитектуры (как на рис. 12-1), можно с помощью экспертов в данной прикладной области приступать к разработке основных сценариев поведения системы, как это было описано в главе 6. Чтобы подробнее описать ожидаемое поведение системы, можно использовать диаграммы взаимодействия, диаграммы объектов, протоколы действий или прототипы. На рис. 12-2 приведена диаграмма взаимодействия компонент системы, отражающая сценарий подготовки ежедневных приказов по движению поездов. На данном уровне анализа нас интересуют именно основные события и взаимодействия, определяющие поведение системы. Такие детали, как сигнатуры операций и ассоциации - это тактические подробности, которые понадобятся на последующих фазах проектирования.

В системе таких размеров запросто можно найти сотни первичных сценариев.[41] В главе 6 мы уже установили "правило 80%". Это значит, что до перехода к про

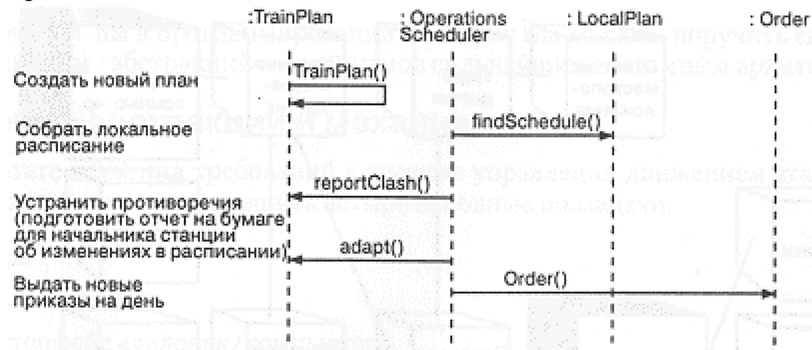


Рис. 12-2. Подготовка ежедневных приказов по движению

ектированию архитектуры желательно зафиксировать 80% важнейших сценариев. Дожидаться 100% готовности бессмысленно.

Очевидно, нужно перевести требования к системе на язык требований к ее программной и аппаратной частям, чтобы различные компетентные организации могли одновременно заниматься отдельными частями задачи (но обязательно под присмотром некоторой центральной группы, обеспечивающей общее видение проекта). Совместное создание аппаратного и программного обеспечения - сложная задача, особенно, если эти части слабо связаны и создаются разными фирмами. Иногда ясно, какая аппаратура будет использоваться. Например, можно использовать готовые терминалы или рабочие станции для бортовых дисплейных систем и в центрах управления операциями. Аналогично, представляется вполне очевидным, что составлением расписаний поездов занимаются программы. Окончательное решение о том, какую основу, аппаратную или программную, использовать в каждом конкретном случае, зависит от предпочтений разработчиков не меньше, чем от всего остального. Специализированную аппаратуру можно использовать, когда важнее производительность, а использование программ целесообразнее, когда необходимо обеспечить гибкость.

Будем считать, что первоначальный вариант аппаратной архитектуры выбран архитекторами системы. Этот выбор не должен считаться окончательным, но по крайней мере он дает отправную точку для уточнения требований к программному обеспечению. В ходе анализа, а затем и проектирования, нам необходима свобода в выборе аппаратной или программной реализации той или иной функции: позднее может оказаться, что нужна дополнительная аппаратура, или что данную функцию можно реализовать программно.

На рис. 12-3 показано целевое аппаратное обеспечение для системы управления движением; здесь используются наши обозначения для диаграмм процессов. Эта архитектура процессов соответствует схеме на рис. 12-1. В частности, предусмотрен один бортовой компьютер на каждом поезде, соединяющий систему сбора и передачи информации о локомотиве, систему управления энергией, бортовой дисплей и устройство управления данными. Мы предполагаем, что некоторые бортовые устройства, такие, как дисплей, обладают минимальным интеллектом, но, возможно, не все они программируемые. Мы полагаем, что каждый ответчик подсоединен к передатчику, который посылает сообщения на проходящий мимо него поезд; компьютер к ответчику местоположения не подсоединен. Все группы путевых устройств (каждое из которых логически состоит из интерфейса и переключателя) управляются компьютером, который может взаимодействовать с проходя-

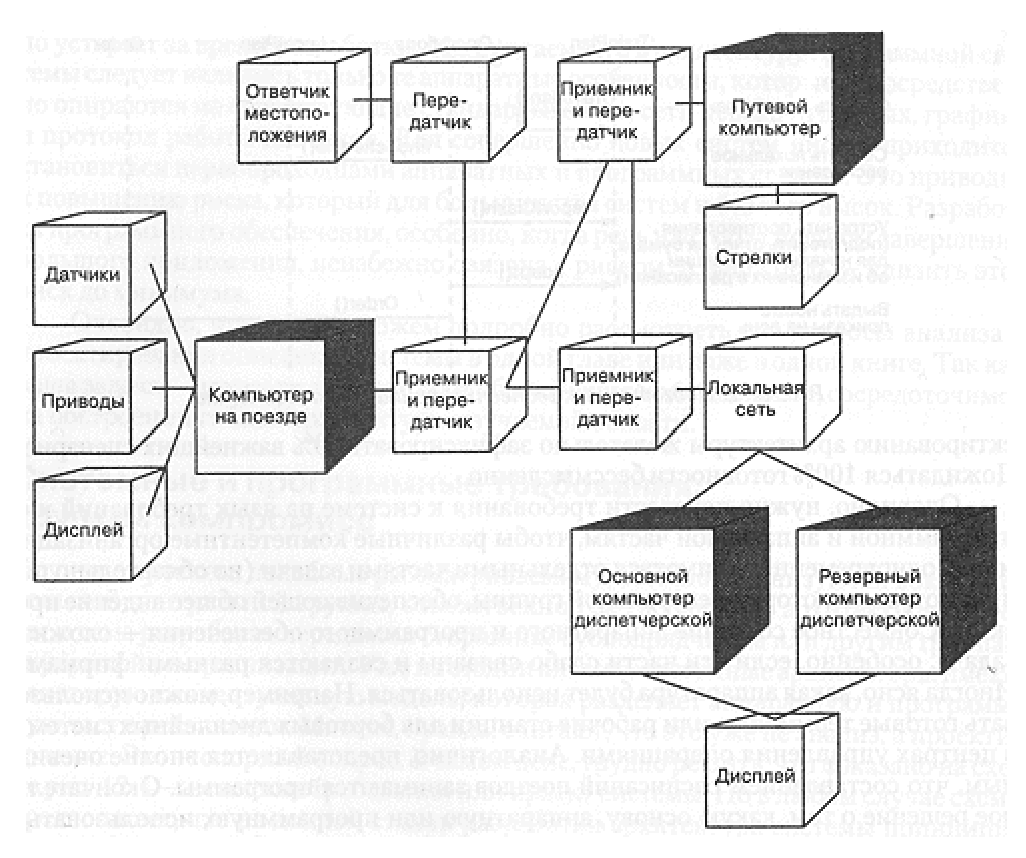


Рис. 12-3. Диаграмма процессов системы управления движением

щим поездом или с наземным контроллером через их передатчики и приемники. Каждый наземный контроллер присоединяется через глобальную сеть к диспетчерскому центру (который входит в систему управления операциями). Для обеспечения бесперебойного обслуживания мы решили разместить на каждом диспетчерском центре два компьютера: основной и резервный (второй включится в случае отказа основного компьютера). В свободное время резервный компьютер может использоваться для обслуживания других, низкоприоритетных пользователей.

На эксплуатационном уровне система управления движением может содержать сотни компьютеров: по одному на каждый поезд, по одному на каждый блок интерфейса путевых устройств и по два на каждый диспетчерский центр. На диаграмме процессов показаны только некоторые компьютеры, так как излишне показывать повторяющиеся компоненты конфигурации.

Как уже говорилось в главах 6 и 7, здравый смысл подсказывает, что при разработке большого проекта огромную роль играют разумность и ясность интерфейсов между ключевыми частями системы. Особенно это важно для интерфейса между программной и аппаратной частями системы. В начале работы над проектом интерфейс может быть определен не полностью, но он должен быть достаточно быстро формализован, чтобы различные части системы можно было разрабатывать, тестировать и интегрировать одновременно. Хорошо определенный интерфейс позволяет производить сборку системы без существенных переделок ее частей. Кроме того, мы не рассчитываем, что все разработчики, участвующие в проекте, будут одинаково сильны в программировании. Поэтому мы должны поручить спецификации ключевых абстракций и механизмов сильнейшим системным архитекторам

Ключевые абстракции и механизмы

В результате изучения требований к системе управления движением становится очевидно, что мы должны решить четыре основные подзадачи:

- сеть

•база данных

•интерфейс "человек/компьютер"

•управление аналоговыми устройствами в реальном времени.

Как мы пришли к выводу, что именно в этих подзадачах сконцентрирован основной риск разработки?

Систему связывает воедино распределенная сеть передачи данных. С помощью радио передаются сообщения: между ответчиками и поездами, между поездами и наземными контроллерами, между поездами и блоками интерфейсов путевых устройств, между наземными контроллерами и путевыми устройствами. Кроме того, сообщения должны передаваться между диспетчерскими центрами и отдельными наземными контроллерами. Надежная работа всей системы обеспечивается своевременным и надежным приемом и передачей сообщений.

Кроме того, система должна одновременно хранить информацию о местоположении и планируемых маршрутах множества поездов. Мы должны поддерживать постоянно обновляемую информацию и гарантировать ее целостность даже в случае попыток одновременно записать и считать информацию из разных мест сети. Следовательно, нам нужна распределенная база данных.

Проектирование человеко-машинного интерфейса ставит еще одну группу задач. Дело в том, что пользователями системы в основном являются машинисты и диспетчеры; но никто из них не обязан обладать профессиональными навыками работы с компьютером. Пользовательский интерфейс операционных систем, таких как UNIX или Windows, пригоден (по большей части) для специалиста-программиста, но считается слишком враждебным для конечных пользователей таких сред, как система управления движением. Следовательно, все формы взаимодействия должны быть спроектированы в расчете на эту особую группу пользователей.

Наконец, система управления движением должна взаимодействовать с разнообразными датчиками и исполнительными механизмами. Не останавливаясь здесь на природе этих устройств, отметим, что принципы управления ими не зависят от конкретного типа устройства и должны быть выбраны однотипными во всей системе.

Каждая из этих четырех подзадач включает целый ряд обособленных вопросов. Системные архитекторы должны найти ключевые абстракции и механизмы каждой задачи, и тогда мы сможем пригласить экспертов для решения каждой отдельной подзадачи независимо от других. Однако, ни анализ, ни проектирование не удастся завершить за один проход, - круг за кругом анализ будет обнаруживать новые архитектурные проблемы, решение которых потребует нового анализа. Таким образом, разработка будет неизбежно пошаговой и итеративной.

Из краткого проблемного анализа четырех главных подзадач мы видим, что существуют три высокоуровневые ключевые абстракции:

• Поезда	Локомотивы и вагоны.
• Пути	Профиль пути, его качество и путевые устройства.
• Планы	Расписания, приказы, устранение накладок, назначение полномочии и подбор бригад.

Каждый поезд характеризуется текущим положением на путях и может иметь только один активный план движения. Аналогично, в каждой точке пути может быть самое большое один поезд. Каждый план относится только к одному поезду, но ко многим точкам пути.

Мы можем выделить ключевой механизм для каждой из четырех (почти независимых) подзадач:

•передача сообщений

- планирование движения поездов
- отображение информации
- сбор данных от датчиков.

Эти четыре механизма составляют душу нашей системы. Они являются наиболее сложными и рискованными частями проекта. Важно, чтобы мы поручили лучшим системным архитекторам поэкспериментировать с различными подходами и постепенно создать среду, на базе которой более молодые разработчики сделают все остальное.

12.2. Проектирование

Как уже отмечалось в главе 6, создание архитектуры подразумевает выявление основной структуры классов и спецификацию общих взаимодействий, которые оживляют классы. Сконцентрировав внимание прежде всего на этих механизмах, мы с самого начала выявляем элементы наибольшего риска и нацеливаем на них все усилия системных архитекторов. Результаты этой фазы дают хорошую основу (в виде классов и взаимодействий), на базе которой строятся функциональные элементы нашей системы.

В данном разделе мы подробно рассмотрим семантику каждого из четырех выделенных ключевых механизмов.

Механизм передачи сообщений

Под сообщением здесь мы не имеем в виду активизацию методов, как это принято в объектно-ориентированных языках программирования. В данном случае понятие взято из словаря предметной области, из самого высокого уровня абстракции. Вот несколько примеров сообщений в системе управления движением: сигнал запуска путевому устройству, сообщение о прохождении поезда через определенный пункт пути, приказ диспетчера машинисту. Все эти виды сообщений могут передаваться внутри системы управления движением на двух уровнях:

- между компьютерами и устройствами
- между компьютерами.

Сейчас нас интересует второй уровень передачи сообщений. Так как система включает территориально распределенную сеть, мы должны учесть такие факторы, как помехи, отказы оборудования и секретность передачи информации.

Первый шаг при определении сообщений в системе - анализ взаимодействия каждой пары сообщающихся компьютеров (см. рис. 12-3). Для каждой такой пары мы должны задать три вопроса: (1) Какую информацию обрабатывает каждый компьютер? (2) Какая информация будет передаваться с одного компьютера на другой? (3) К какому уровню абстракции будет относиться эта информация? Эмпирического ответа на эти вопросы нет. Мы должны действовать итеративно, пока не придем к уверенности, что определены правильные сообщения и в системе связи нет "узких" мест (которые могут возникать из-за перегрузки линий связи или, например, из-за того, что сообщение разбивается на слишком мелкие пакеты).

Очень важно, чтобы на данном этапе проектирования внимание было сосредоточено на сути, а не на форме сообщений. Слишком часто системные архитекторы начинают проектирование с выбора битового представления сообщений. В реальной задаче преждевременный выбор низкоуровневого представления обязательно приведет к изменениям в дальнейшем и затронет всех, кто пользовался этим представлением. Кроме того, на ранней стадии проектирования у нас пока нет полной информации, как будут использоваться

данные сообщения, и, следовательно, мы не можем судить, какое представление будет оптимальным по размеру и времени передачи.

Концентрируя внимание на сути сообщений, мы рассмотрим все классы сообщений. Другими словами, нужно определить назначение и смысл каждого сообщения, а также перечислить операции их обработки.

На диаграмме классов на рис. 12-4 показаны некоторые наиболее важные сообщения в системе управления движением. Заметим, что все сообщения в конечном счете являются экземплярами абстрактного класса **Message**, который инкап-

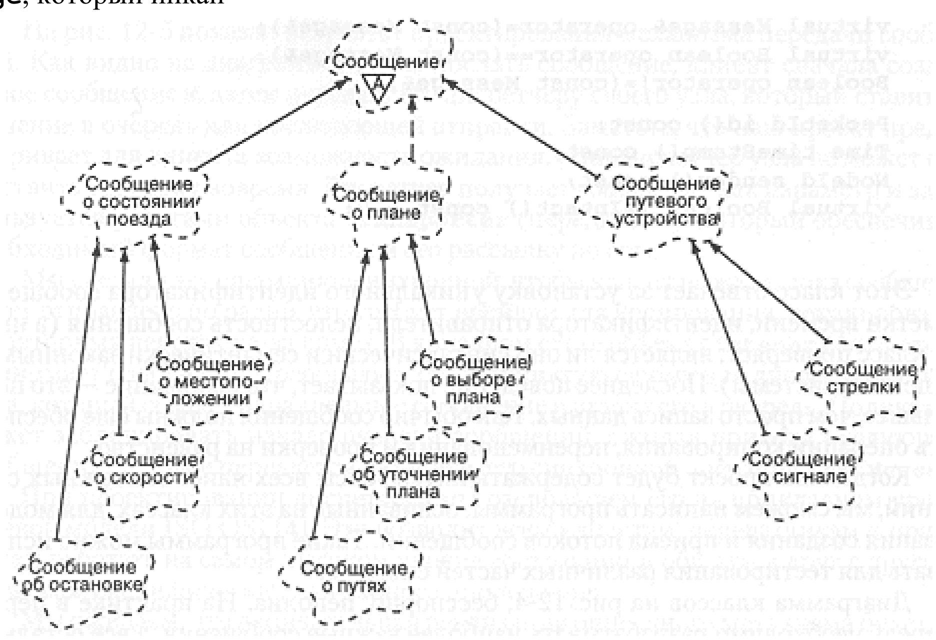


Рис. 12-4. Диаграмма классов сообщений

сулирует поведение, общее для всех сообщений. Три класса следующего уровня представляют главные категории сообщений: сообщение о состоянии поезда, сообщение о плане движения поезда, сообщение путевого устройства. Каждый из этих трех классов будет детализирован в дальнейшем. В результате проектирования должны появиться десятки специализированных классов. Таким образом, существование обобщающих абстрактных классов чрезвычайно важно; без них мы получили бы сотни несвязанных между собой и, следовательно, сложных в использовании модулей, каждый из которых реализовывал бы специализированный класс, По мере проектирования мы будем выявлять другие важные группы сообщений и создавать для них специализированные промежуточные классы. К счастью, изменения в иерархии классов не должны волновать клиентов, использующих классы.

Прежде всего нам следует стабилизировать интерфейсы ключевых классов сообщений. Начинать этот процесс лучше всего с основных классов иерархии. Начнем с введения двух следующих типов;

```
//номер, обозначающий уникальный идентификатор пакета
typedef unsigned int Packetid;
//номер, обозначающий уникальный сетевой идентификатор
typedet unsigned Int NodeId;
```

Теперь дадим определение абстрактного класса **Message**:

```
class Message {
public:
      Message () ;
      Message(NodeId sender);
      Message(const Message&);
      virtual `Message () ;
      virtual Message& operator=(const Message&);
      virtual Boolean operator==(const Message&);
```

```
        Boolean operator!=(const Message&);
        PacketId id() const;
        Time timeStamp() const;
        NodeId sender() const;
        virtual Boolean isIntact() const = 0;
};
```

Этот класс отвечает за установку уникального идентификатора сообщения, отметки времени, идентификатора отправителя, целостность сообщения (а именно, класс проверяет, является ли оно синтаксически и семантически законным сообщением системы). Последнее поведение показывает, что сообщение - это нечто большее, чем просто запись данных. Как обычно сообщения должны еще обеспечивать операции копирования, переименования и проверки на равенство.

Когда наш проект будет содержать интерфейсы всех наиболее важных сообщений. мы сможем написать программы, основанные на этих классах, для моделирования создания и приема потоков сообщений. Такие программы можно использовать для тестирования различных частей системы.

Диаграмма классов на рис. 12-4, бесспорно, неполна. На практике в первую очередь необходимо разрабатывать наиболее важные сообщения, а все остальные добавлять по мере того, как будут обнаруживаться менее общие формы взаимодей-

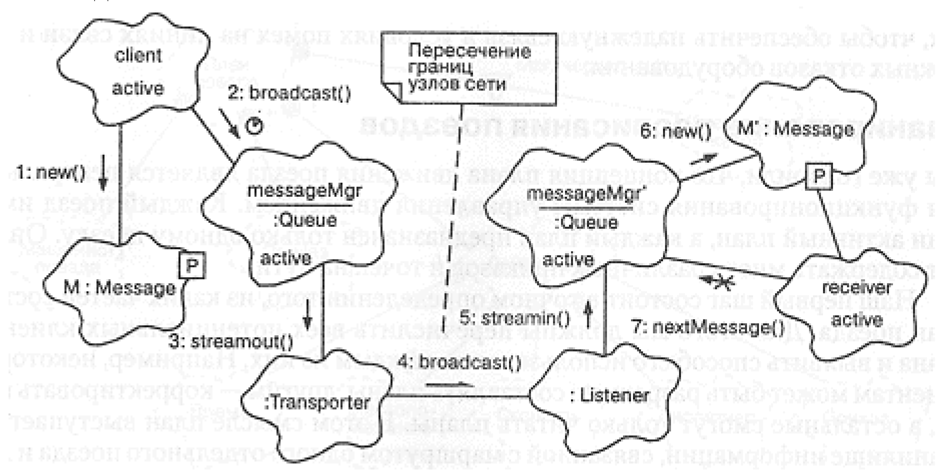


Рис. 12-5. Передача сообщений

ствия. Использование объектно-ориентированного проектирования позволит нам последовательно добавлять эти сообщения без нарушения существующих частей системы, так как возможность изменений учтена с самого начала.

Если мы удовлетворены структурой классов, то можно начать проектирование самого механизма передачи сообщений. Здесь возникают две конкурирующих между собой цели: придумать механизм, который обеспечит надежную доставку сообщении, но сделает это на достаточно высоком уровне абстракции, так, чтобы клиенту не надо было заботиться о способе доставки сообщения. Такой механизм передачи сообщений позволит клиентам ограничиться упрощенным представлением о процессе передачи.

На рис. 12-5 показан результат проектирования механизма передачи сообщений. Как видно на диаграмме, чтобы послать сообщение, клиент сначала создает новое сообщение м, затем передает его диспетчеру своего узла, который ставит сообщение в очередь для последующей отправки. Заметьте, что наш проект предусматривает для клиента возможность ожидания, если диспетчер узла не может осуществить отправку вовремя. Диспетчер получает сообщение как параметр и затем пользуется услугами объекта **`Transporter`** (передатчик), который обеспечивает необходимый формат сообщения и его рассылку по сети.

Мы сделали эту операцию асинхронной, чтобы клиент не ждал, пока сообщение будет отправлено по радио, что требует времени для кодирования, декодирования и повторных передач из-за помех. В конечном счете объект **`Listener`** (слушатель) принимает сообщение, преобразует его в принятую форму для диспетчера своего узла, который создает параллельное сообщение и ставит его в очередь. Получатель может заблокировать начало очереди сообщений, ожидая прихода следующего сообщения, которое передается как параметр синхронной операции **`nextMessage`**.

При проектировании диспетчера мы располагаем его на прикладном уровне сетевой модели ISO OSI [4]. Это позволит всем клиентам, передатчикам и приемникам работать на самом высоком уровне абстракции и общаться друг с другом в терминах, специфических для данного приложения.

Мы ожидаем, что окончательная реализация описанного механизма будет, вероятно, несколько более сложной. Например, может потребоваться шифровать и дешифровать сообщение и использовать коды обнаружения и исправления ошибок, чтобы обеспечить надежную связь в условиях помех на линиях связи и возможных отказов оборудования.

Планирование расписания поездов

Мы уже говорили, что концепция плана движения поезда является центральной для функционирования системы управления движением. Каждый поезд имеет один активный план, а каждый план предназначен только одному поезду. Он может содержать много различных приказов и точек на пути.

Наш первый шаг состоит в точном определении того, из каких частей состоит план поезда. Для этого мы должны перечислить всех потенциальных клиентов плана и выявить способ его использования каждым из них. Например, некоторым клиентам может быть разрешено составлять планы, другим - корректировать планы, а остальные смогут только читать планы. В этом смысле план выступает как хранилище информации, связанной с маршрутом одного отдельного поезда и действиями во время движения. Примером таких действий может быть отцепление или подцепление вагонов.

На рис. 12-6 приведены стратегические проектные решения, касающиеся структуры класса **`TrainPlan`**. Как и в главе 10, мы используем диаграмму классов, чтобы показать части, из которых состоит план движения поезда (подобно тому, как это делается на традиционных диаграммах "сущность-связь"). Мы видим, что каждый план содержит одну бригаду, но может включать в себя много приказов и действий. Мы ожидаем, что эти действия будут упорядочены во времени и что с каждым действием связана такая информация, как время, местоположение, скорость, ответственное лицо, приказы. Например, план может содержать следующие действия:

Время	Положение	Скорость	Ответственное лицо	Приказ
0800	Pueblo	Как указано	Начальник депо	Покинуть депо
1100	Colorado Springs	40 миль/ч		Отцепить 30 вагонов
1300	Denver	40 киль/ч		Отцепить 20 вагонов
1600	Pueblo	Как указано		Вернуться в депо

Из рис. 12-6 видно, что класс **`TrainPlan`** имеет один статический объект типа **`UniqueId`**, так называемое магическое число, однозначно идентифицирующее каждый экземпляр класса **`TrainPlan`**.

Как это делалось для класса **`Message`** и его подклассов, можно в первую очередь спроектировать наиболее важные элементы плана движения

поезда; детали будут проясняться по мере того, как мы будем использовать план для разных клиентов.

Одновременное наличие огромного числа активных и неактивных планов поездов возвращает нас к проблеме базы данных, о которой мы уже говорили. Диаграмма классов на рис. 12-6 может служить наброском логической схемы этой базы данных. При этом возникает следующий вопрос: где хранится план поезда?

В совершенном мире, где нет помех или задержек при передаче и где неограничены ресурсы компьютеров, лучше всего было бы разместить все планы движения поездов в единой центральной базе данных. Такой подход обеспечивает существование единственного экземпляра каждого плана. Однако реальные условия делают это решение неэффективным: неизбежны задержки при передаче, производительность процессоров ограничена. Таким образом, скорость доступа к плану, который

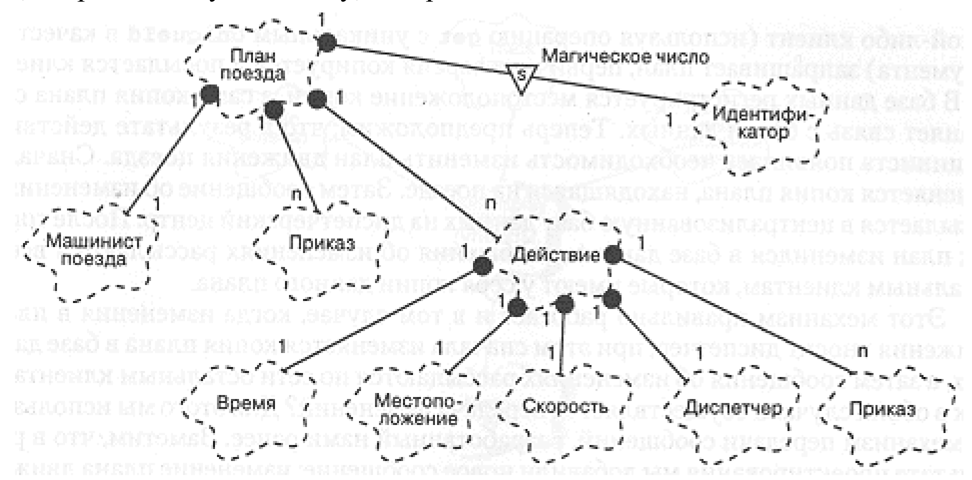


Рис. 12-6. Диаграмма классов **`TrainPlan`** *(план движения поезда)*

расположен в диспетчерском центре, с поезда, не будет отвечать требованиям реального времени. Однако, с помощью программного обеспечения можно создать иллюзию централизованной базы данных. Наше решение заключается в том, что планы поездов будут располагаться на компьютерах диспетчерского центра, а копии этих планов будут по мере необходимости распределяться по узлам сети. Для обеспечения эффективности компьютер каждого поезда может хранить копию своего плана. Таким образом, бортовое программное обеспечение может получить нужные сведения с пренебрежимо малой задержкой. Если план изменяется в результате действий диспетчера или (что менее вероятно) по решению машиниста, наше программное обеспечение должно гарантировать, что все копии этого плана обновятся, причем за разумное время.

На рис. 12-7 показано, как происходит передача и обновление копий плана. Первичная копия плана движения находится в централизованной базе данных в диспетчерском центре и может быть разослана по любому числу узлов сети. Когда

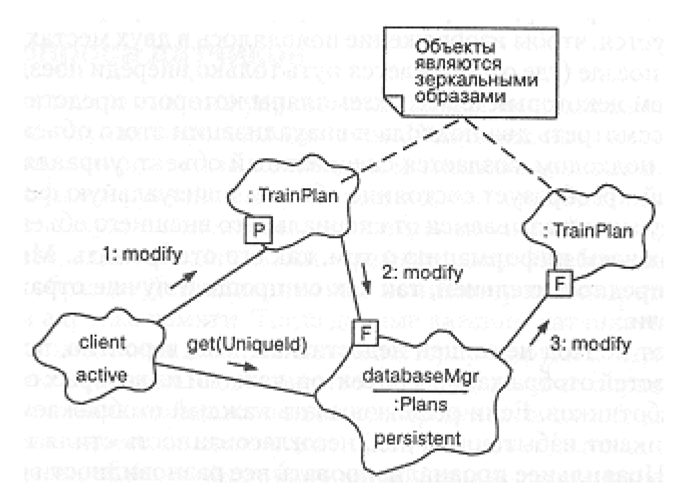


Рис. 12- 7. План движения поезда

какой-либо клиент (используя операцию **get** с уникальным **UniqueId** в качестве аргумента) запрашивает план, первичная версия копируется и посылается клиенту. В базе данных регистрируется местоположение копии, а сама копия плана сохраняет связь с базой данных. Теперь предположим, что в результате действий машиниста появилась необходимость изменить план движения поезда. Сначала изменяется копия плана, находящаяся на поезде. Затем сообщение об изменениях посылается в централизованную базу данных на диспетчерский центр. После того, как план изменился в базе данных, сообщения об изменениях рассылаются всем остальным клиентам, которые имеют у себя копии данного плана.

Этот механизм правильно работает и в том случае, когда изменения в план движения вносит диспетчер; при этом сначала изменяется копия плана в базе данных, а затем сообщения об изменениях рассылаются но сети остальным клиентам. Как в обоих случаях осуществляется передача изменении? Для этого мы используем механизм передачи сообщений, разработанный нами ранее. Заметим, что в результате проектирования мы добавили новое сообщение: изменение плана движения поезда. Таким образом, механизм передачи планов движения базируется на уже существующем низкоуровневом механизме передачи сообщений.

Использование готовой коммерческой СУБД на диспетчерских компьютерах позволит обеспечить резервирование данных, восстановление, ведение контрольного журнала и секретность информации.

Отображение информации

Использование готовых технологических решений для базы данных позволяет нам сосредоточиться на специфике задачи. Такого же результата можно добиться и в механизмах отображения информации, если использовать стандартные графические средства, например, Microsoft Windows или X Windows. Использование готовых графических программных средств поднимает уровень абстракции нашей системы настолько, что разработчикам не надо беспокоиться об отображении информации на уровне пикселей. Кроме того, очень важно инкапсулировать проектные решения о графическом представлении различных объектов.

Рассмотрим, например, отображение информации о профиле и качестве участков пути. Требуется, чтобы изображение появлялось в двух местах: в диспетчерском центре и на поезде (где отображается путь только впереди поезда). Предполагая, что мы имеем некоторый класс, экземпляры которого представляют участки пути, можно рассмотреть два подхода к визуализации этого объекта. В соответствии с первым подходом, создается специальный объект, управляющий отображением, который преобразует

состояние объекта в визуальную форму. Согласно второму подходу мы отказываемся от специального внешнего объекта и в каждый наш объект включаем информацию о том, как его отображать. Мы считаем, что второй подход предпочтительней, так как он проще и лучше отражает сущность объектной модели.

Однако, этот подход не лишен недостатков. Мы, вероятно, получим множество разновидностей отображаемых объектов, каждый из которых создан разными группами разработчиков. Если реализовывать каждый отображаемый объект отдельно, то возникают избыточный код. несогласованность стиля и вообще большая путаница. Правильнее проанализировать все разновидности отображаемых объектов, определить, какие у них общие элементы и создать набор промежуточных классов, который обеспечит отображение этих общих элементов. В свою оче-

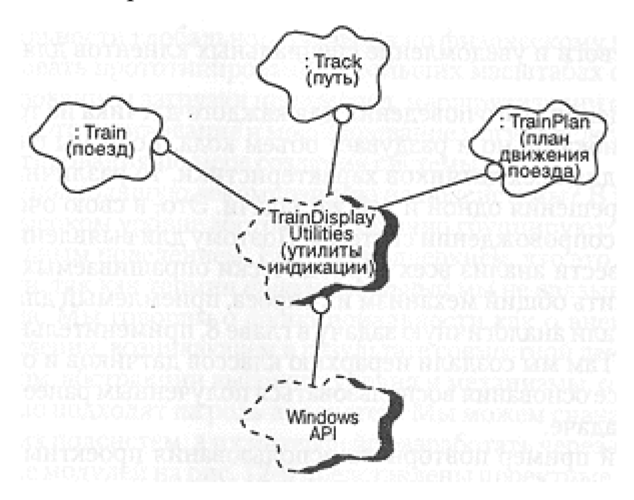


Рис. 12-8. Отображение информации

редь, промежуточные классы могут быть построены на основе коммерческих низкоуровневых графических пакетов.

На рис. 12-8 показано проектное решение о реализации всех отображаемых объектов с помощью общих утилит класса. Эти утилиты построены на основе низкоуровневого интерфейса Windows, который скрыт от всех высокоуровневых классов. На самом деле, процедуры Windows API трудно воплотить в одном классе или утилите. Наша диаграмма немного упрощена; вероятно, реализация потребует услуг нескольких классов Windows API и утилит отображения на дисплее компьютера в поезде.

Основное достоинство предлагаемого подхода заключается в том, что уменьшается влияние изменений, возникающих при перераспределении роли аппаратуры и программ. Например, если нам надо заменить наши дисплеи на более (менее) мощные, придется подправить процедуры только в классе **`TrainDisplayUtilities`**. Без такой декомпозиции нам бы пришлось вносить изменения в каждый отображаемый объект при любых изменениях на нижнем уровне.

Механизм опроса датчиков

Выше мы говорили, что система управления движением должна включать в себя большое количество разнообразных датчиков. Например, на каждом поезде датчики следят за температурой масла, количеством топлива, дроссельной установкой, температурой воды, нагрузкой на двигатель и т. д. Активные датчики путевых устройств сообщают текущее положение своих переключателей и передают сигналы. Все значения, возвращаемые датчиками - разные, но их обработка может производиться сходным образом. Допустим, что наш компьютер использует ввод-вывод по фиксированным адресам памяти. Тогда данные каждого датчика читаются из определенной области

памяти и только потом интерпретируются способом, зависящим от конкретного датчика. Большинство датчиков должно опрашиваться периодически. Если значение находится в заданных пределах, оно сообщается какому-то клиенту, и больше ничего не происходит. Если же отсчет датчика вышел за установленные пределы, об этом могут быть оповещены и другие клиенты. Наконец, если отсчет вышел далеко за допустимые границы (например, давление масла на локомотиве поднимается до опасного уровня), может понадобиться какой-то звуковой сигнал тревоги и уведомление специальных клиентов для принятия решительных мер.

Воспроизведение этого поведения для каждого датчика не только утомительно и чревато ошибками, но и раздувает объем кода. Если мы с самого начала не выделим общие для всех датчиков характеристики, то различные разработчики предложат свои решения одной и той же задачи. Это, в свою очередь, приведет к сложностям при сопровождении системы. Поэтому для выявления общих свойств необходимо провести анализ всех периодически опрашиваемых аналоговых датчиков и предложить общий механизм их опроса, приемлемый для всех.

Мы уже решали аналогичную задачу в главе 8, применительно к метеорологической станции. Там мы создали иерархию классов датчиков и описали механизм их опроса. Есть все основания воспользоваться полученным ранее решением и в нашей нынешней задаче.

Это хороший пример повторного использования проектных решении в различных прикладных областях.

12.3. Эволюция

Модульная архитектура

Мы уже говорили о том, что модульность для больших систем необходима, но не достаточна; для задач такого масштаба, как система управления движением, нужно сосредоточиться на декомпозиции по подсистемам. На ранних стадиях эволюции мы должны разработать модульную архитектуру системы, представляющую физическую структуру ее программного обеспечения.

Проектирование программного обеспечения для очень больших систем должно начинаться до полного завершения проектирования аппаратных средств. Написание программы занимает, как правило, даже больше времени, чем разработка аппаратуры. Кроме того, по ходу процесса функциональность может перераспределяться между аппаратной и программной частями. Поэтому зависимость от аппаратуры должна быть максимально изолирована, так, чтобы программные средства можно было начать проектировать без привязки к аппаратуре. Это означает также, что разработка должна основываться на идее взаимозаменяемых подсистем. В системах управления и контроля, таких, как система управления движением, нужно сохранить возможность задействовать новые аппаратные решения, которые могут появиться в процессе разработки программного обеспечения.

На ранних этапах мы должны разумно провести декомпозицию программного обеспечения, чтобы субподрядчики, ответственные за различные части системы, могли работать одновременно (возможно даже используя различные языки программирования). Как уже говорилось в главе 7, существует много причин нетехнического характера, определяющих физическую декомпозицию больших систем. Наиболее важен вопрос взаимодействия различных групп разработчиков. Отношения между субподрядчиками складываются обычно на достаточно ранних стадиях жизни системы, часто до получения информации, достаточной для выбора правильной декомпозиции системы.

Желательно, чтобы системные архитекторы поэкспериментировали с несколькими альтернативными декомпозициями на подсистемы для того, чтобы быть уверенными в правильности глобального решения по физическому проектированию. Можно задействовать прототипирование в больших масштабах с имитацией подсистем и моделированием загрузки процессора, маршрутизации сообщений и внешних событий. Прототипирование и моделирование могут послужить основой для нисходящего тестирования по мере создания системы.

Как выбрать подходящую декомпозицию на подсистемы? В главе 4 отмечено, что объекты на высоком уровне абстракции обычно группируются в соответствии с их функциональным повелением. Еще раз подчеркнем, что это не противоречит объектной модели, так как термин функциональный мы не связываем жестко с понятием алгоритма. Мы говорим о функциональности как о внешнем видимом и тестируемом поведении, возникающем в результате совместной деятельности объектов. Таким образом, абстракции высокого уровня и механизмы, о которых говорилось ранее, хорошо подходят на роль подсистем. Мы можем сначала допустить существование таких подсистем, а их интерфейс разработать через некоторое время.

На диаграмме модулей на рис. 12-9 представлены проектные решения верхнего уровня модульной архитектуры системы управления движением. Каждый уровень здесь соответствует выделенным ранее четырем подзадачам: сеть передачи данных, база данных, аналоговые устройства управления в реальном времени, интерфейс "человек/компьютер".

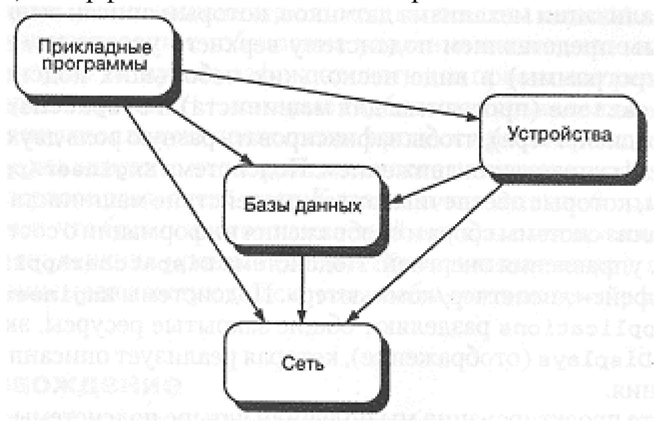


Рис. 12-9. Диаграмма модулей верхнего уровня системы управления движением

Спецификация подсистем

Если мы рассмотрим внешнее представление любой из подсистем, то обнаружим, что она обладает всеми характеристиками объекта. Каждая подсистема имеет уникальную, хотя и статичную, идентичность и большое число возможных состояний, а также демонстрирует очень сложное поведение. Подсистемы используются как хранилища других классов, утилит классов и объектов; таким образом, они лучше всего характеризуются экспортируемыми ресурсами. На практике при использовании C++ эти ресурсы представляются в форме каталогов, содержащих логически связанные модули и вложенные подсистемы.

Диаграмма модулей на рис. 12-9 полезна, но не полна, так как каждая из подсистем на ней слишком велика для реализации одним небольшим коллективом разработчиков. Мы должны раскрыть внутреннее представление подсистем верхнего уровня и провести их декомпозицию.

Рассмотрим для примера подсистему **`NetworkFacilities`** (сеть). Мы решили разбить ее на две другие подсистемы, одна из которых - закрытая (**`RadioCommunication`** (радиосвязь)), а другая - открытая (**`Messages`** (сообщения)). Закрытая подсистема скрывает детали своего программного

управления физическими устройствами, в то время как открытая подсистема обеспечивает поддержку спроектированного ранее механизма передачи сообщений.

Подсистема, названная **Databases** (базы данных), построена на основе ресурсов подсистемы **NetworkFacilities** и служит для реализации механизма планов движения поезда, который мы описали выше. Мы составляем эту подсистему из двух экспортируемых открытых подсистем, **TrainPlanDatabase** (база данных планов поездов) и **TrackDatabase** (база данных путей). Для действий, общих для этих двух подсистем, мы предусмотрим закрытую подсистему **DatabaseManager** (менеджер баз данных).

Подсистема **Devices** (устройства) также естественно разбивается на несколько небольших подсистем. Мы решили сгруппировать программы, относящиеся ко всем путевым устройствам, в одну подсистему, а программы, связанные с активными механизмами и датчиками локомотива, - в другую. Эти две подсистемы доступны клиентам подсистемы **Devices**, и обе они построены на основе ресурсов подсистем **TrainPlanDatabase** и **Messages**. Таким образом, мы спроектировали подсистему **Devices** для реализации механизма датчиков, который описан выше.

Наконец, мы представляем подсистему верхнего уровня **UserApplications** (прикладные программы) в виде нескольких небольших подсистем, включая **EngineerApplications** (программы для машиниста) и **DispatcherApplications** (программы для диспетчера), чтобы зафиксировать разную роль двух главных пользователей системы управления движением. Подсистема **EngineerApplications** содержит ресурсы, которые обеспечивают взаимодействие машиниста и компьютера, в частности, анализ системы сбора и отображения информации о состоянии локомотива и системы управления энергией. Подсистема **DispatcherApplicatlona** обеспечивает интерфейс "диспетчер/компьютер". Подсистемы **EngineerApplications** и **DispatcherApplications** разделяют общие закрытые ресурсы, экспортируемые из подсистемы **Displays** (отображение), которая реализует описанный ранее механизм отображения.

В результате проектирования мы получили четыре подсистемы верхнего уровня и десять подсистем следующего уровня, в которых размещены все введенные ранее ключевые абстракции и механизмы. Важно, что в терминах этих подсистем можно планировать работу, управлять конфигурациями и версиями. Как говорилось в главе 7, отвечать за каждую такую подсистему может один человек, в то время как разрабатывать ее будет множество программистов. Ответственный за подсистему детализирует ее проект и реализацию и управляет ее интерфейсом с другими подсистемами на том же уровне абстракции. Так, за счет декомпозиции сложной задачи на несколько более простых, становится возможным управление разработкой сложных проектов.

В главе 7 уже демонстрировалась возможность нескольких одновременных представлений разрабатываемой системы. Набор совместимых версий подсистем образует релиз, и таких релиэов может быть множество - по одному на каждого разработчика, еще один - для тестирования, один - для опробования пользователями и т. д. Отдельные проектировщики могут для своих нужд создавать собственные стабильные релизы и интегрировать в них те части, за которые они отвечают, до передачи их остальным. Так создается механизм непрерывной интеграции нового кода.

Основой успеха является тщательное конструирование интерфейсов подсистем. После того как интерфейсы определены, они должны тщательно оберегаться. Как мы определяем внешнее представление подсистемы? Нужно каждую подсистему рассматривать как объект. Поэтому мы ставим те же вопросы, которые задавали в главе 4 для значительно более простых объектов:

Какие состояния имеет объект? Какие действия над ним может выполнить клиент? Каких действий он требует от других объектов?

Например, рассмотрим подсистему **TrainPlanDatabase**. Она строится на основе трех других подсистем (**Messages**, **TrainDatabase**, **TrackDatabase**) и имеет нескольких важных клиентов - подсистемы **WaysideDevices** (путевые устройства), **LocomotiveDevices** (устройства на локомотиве), **EngineerApplications** и **DispatcherApplications**. Подсистема **TrainPlanBatabase** относительно проста - она содержит все планы поездов. Конечно, хитрость в том, что эта подсистема должна поддерживать механизм распределенной передачи планов движением поезда. Снаружи клиент видит монолитную базу данных, но изнутри мы знаем, что на самом деле база данных - распределенная, и поэтому должны основывать се на механизме передачи сообщений подсистемы **Messages**.

Какие действия можно выполнять с помощью **TrainPlanDatabase**? Все обычные для базы данных операции: добавление, удаление и изменение записей, запросы. Так же как в главе 10, нужно зафиксировать все проектные решения об этой подсистеме в форме классов C++, которые снабдят нас объявлениями операций.

На этой стадии нам следует продолжить процесс проектирования для каждой подсистемы. Еще раз отметим, что вероятность того, что все интерфейсы окажутся правильными с первого раза, очень мала. К счастью, как и для небольших объектов, опыт подсказывает, что большинство изменений, которые мы произведем в интерфейсах, не затронет верхних уровней (совместимость снизу вверх), если мы хорошо поработали, описывая каждую подсистему в объектно-ориентированном стиле.

12.4. Сопровождение

Добавление новых функций

Программное обеспечение сопровождается и постоянно дорабатывается, что особенно справедливо для таких больших систем, как наша. Действительно, до сих пор можно встретить программы, разработанные лет двадцать назад (просто патриархальные по компьютерным меркам). Чем больше пользователей применяет систему управления движением и чем лучше мы адаптируем проект к новым требованиям, тем чаще клиенты будут находить новые неожиданные применения для существующих механизмов, создавая потребность во включении в систему новых функций.

Рассмотрим единственное добавление к нашим требованиям: обработку платежной ведомости. Предположим, анализ показал, что работа с платежными ведомостями железнодорожной компании осуществляется с использованием аппаратуры, выпуск которой прекращен, поэтому возник серьезный риск безвозвратной потери всей системы платежей в результате нескольких критических поломок. В этом случае можно объединить обработку платежной ведомости с системой управления движением. Для начала надо понять, как эти две несвязанные задачи будут сосуществовать; можно рассматривать их как разные приложения, причем обработка платежной ведомости будет происходить в фоновом режиме.

Дальнейший анализ показывает, что от интеграции обработки платежной ведомости может быть получена огромная польза. Вспомним, что планы поездов содержат информацию о распределении бригад. Следовательно, мы можем проанализировать запланированное и действительное распределение бригад, вычислить рабочее время, сверхурочные часы и т. п. Получая эту информацию непосредственно, мы можем обрабатывать платежную ведомость дешевле и быстрее.

Что добавление этой функции затрагивает в нашем проекте? Очень немногое. Новую подсистему можно добавить в подсистему `UserApplications`. Оттуда новой подсистеме будут видны все важные механизмы, которые нужны для ее функционирования. Признак хорошо спроектированной объектно-ориентированной системы: значительные дополнения к требованиям могут быть учтены довольно просто путем надстройки новых функций над существующими механизмами.

Предположим, мы хотим ввести более существенное изменение: добавить экспертную систему, помогающую диспетчеру при определении маршрутов и реагирующую на чрезвычайные ситуации. Как это требование отразится на нашем проекте? Незначительно. Мы можем разместить новую подсистему между подсистемами `TrainPlanDatabase` и `DispatcherApplications`, так как база знаний, созданная для экспертной системы, подобна по содержанию `TrainPlanDatabase`; кроме того, подсистема `DispatcherApplications` является единственным клиентом экспертной системы. Нам предстоит разработать некоторый новый механизм, чтобы доводить рекомендации до конечного пользователя. Например, мы можем использовать метафору информационной доски, как это делалось в главе 11.

Изменение аппаратных средств

Мы уже говорили, что аппаратные средства развиваются быстрее, чем программное обеспечение. Более того, всегда будут причины, вынуждающие нас выбрать в ходе проектирования такие аппаратные решения, о которых потом мы будем сожалеть.[42] Поэтому рабочая аппаратура в больших системах устаревает гораздо раньше программы. Например, после нескольких лет эксплуатации мы можем заменить дисплеи на всех поездах и во всех диспетчерских центрах. Как это может повлиять на существующий проект? Если во время разработки мы сохраняли интерфейсы подсистем па высоком уровне абстракции, это изменение аппаратуры приведет лишь к незначительным изменениям в программе. Мы подправим только совокупность процедур, относящуюся к дисплеям, не затрагивая другие подсистемы, которые вообще ничего не знают об особенностях конкретных рабочих станций. Это достигается благодаря тому, что поведение всех рабочих станций скрыто в подсистеме `Displays`. Таким образом, подсистема действует как стена абстракций, которая защищает остальных клиентов от наших трудностей, вызванных разнообразием дисплеев.

Аналогично, радикальные изменения в стандартах телекоммуникации затронут нашу реализацию в очень ограниченном отношении. Наш проект гарантирует, что только подсистема `Massages` связана с сетевыми коммуникациями. Таким образом, фундаментальные изменения в сети не отразятся ни на каком высокоуровневом клиенте; подсистема `Messages` защищает их от капризов сетевой моды.

Итак, воображаемые изменения, которые мы вводили, не смогли разрушить структуру созданного нами проекта. Это - верный признак хорошо спроектированной объектно-ориентированной системы.

Дополнительная литература

Требования к системе управления движением основываются на Продвинутой системе управления поездами (Advanced Train Control System), описанной Марфи (Murphy) [C 1988].

Передача и проверка сообщений присутствует практически во всех системах управления и контроля. Плинта, Ли и Риссман (Plinta, Lee, and Rissman) [C 1989] дали блестящее изложение этих вопросов и предложили

механизм передачи сообщений по процессорам в распределенной системе, безопасный с точки зрения типов.

Ибо книги лишь до некоторой степени
рождаются в мозгах и печенках их авторов.
Большая часть их приходит откуда-то еще,
так что мы, авторы, просто сидим у
пишущих машинок, ожидая, когда книга
случится.

ГИ ЛЕФРАНСУА(GUY
LEFRANCOIS)
Дети (Of Children)

Послесловие

Объектно-ориентированное проектирование - проверенная технология. Наш метод успешно использовался для создания множества сложных систем в самых разных областях.

Потребность в сложных программных системах растет с ошеломляющей быстротой. По мере того, как увеличивается производительность аппаратуры и все больше людей узнает о возможностях компьютеров, нам хочется автоматизировать все более сложные процессы. Фундаментальная ценность объектно-ориентированного проектирования как устоявшейся технологии в том, что оно позволяет человеческому духу сосредоточиться на решении истинно творческих задач при создании сложных систем.

Приложение

Объектно-ориентированные языки программирования

Использование объектно-ориентированной методологии не ограничено каким-либо одним языком программирования - она применима к широкому спектру объектных и объектно-ориентированных языков. Наряду с анализом и проектированием, несомненно важны особенности конкретного языка программирования, поскольку в конечном счете наши конструкции должны быть выражены на каком-то языке. Как отметил Вульф, язык программирования служит трем целям:

• это инструмент проектирования
• это средство человеческого восприятия
• это средство управления компьютером [1].

Данное приложение предназначено для читателей, не знакомых с языками программирования, упоминавшимися в этой книге. Мы приводим сводное описание наиболее важных из них, а также примеры, позволяющие сопоставить синтаксис, семантику и идиомы двух самых интересных - C++ и Smalltalk.

A.1. Концепции

В настоящее время насчитывается более двух тысяч языков программирования высокого уровня. Большинство этих языков возникло исходя из конкретных требований некоторой предметной области. Каждый новый язык позволял перехо-

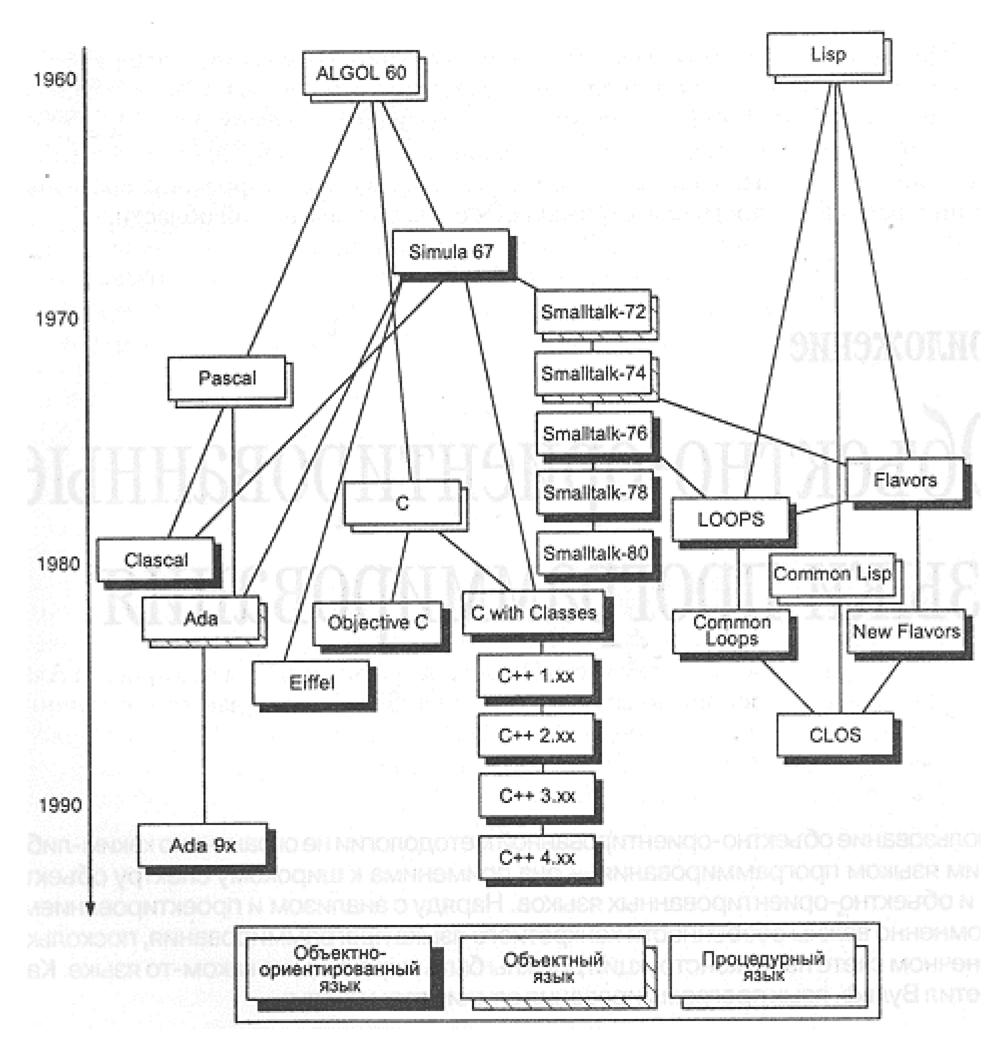


Рис. А-1. Генеалогия объектных и объектно-ориентированных языков

дить ко все более и более сложным задачам. На каждом новом приложении разработчики языков что-то открывали для себя и изменяли свои представления о существенном и несущественном в языке. На развитие языков программирования значительное влияние оказали достижения теории вычислении, которые привели к формальному пониманию семантики операторов, модулей, абстрактных типов данных и процедур.

В главе 2 языки программирования были сгруппированы в четыре поколения по признаку поддерживаемых ими абстракции: математические, алгоритмические, ориентированные на данные, объектно-ориентированные. Самые последние достижения в области развития языков программирования связаны с объектной моделью. К настоящему времени мы насчитали более сотни различных объектных и объектно-ориентированных языков. Как говорилось в главе 2, объектными принято называть языки, которые поддерживают абстракции данных и классы; объектно-ориентированными являются те объектные языки, которые поддерживают наследование и полиморфизм.

Общим предком практически всех используемых сегодня объектных и объектно-ориентированных языков является язык Simula, созданный в 1960 году Далем, Мюрхогом и Ныгардом [2]. Язык Simula основывался на идеях ALGOL, но был дополнен механизмом наследования и инкапсуляции. Но еще более существенно то, что Simula, предназначенная для описания систем и моделирования, ввела дисциплину написания программ, отражающих словарь предметной области.

Рис. А-1, заимствованный у Шмукера [3], демонстрирует генеалогию пяти наиболее влиятельных и популярных объектных или объектно-

ориентированных языков программирования: Smalltalk. Object Pascal, C++, CLOS и Ada. В следующих разделах мы проанализируем некоторые из этих языков с точки зрения их "объектности".

A.2. Smalltalk

Происхождение

Язык Smalltalk был разработан командой Xerox Palo Alto Research Center Learning Research Group (Xerox, Пало Альто, Исследовательский центр, группа исследования обучения), как программная часть Dynabook - фантастического проекта Алана Кея (Alan Kay). В основу были положены идеи Simula, хотя известное влияние оказали также язык FLEX и работы Сеймора Паперта (Seymore Papert) и Валласа Феурзейга (Wallace Feurzeig). Smalltalk является одновременно и языком программирования, и средой разработки программ. Это - чисто объектно-ориентированный язык, в котором абсолютно все рассматривается как объекты; даже целые числа - это классы. Вслед за Simula, Smalltalk является важнейшим объектно-ориентированным языком, поскольку он не только оказал влияние на последующие поколения языков программирования, но и заложил основы современного графического интерфейса пользователя, на которых непосредственно базируются интерфейсы Macintosh, Windows и Motif.

Развитие Smalltalk потребовало почти десятилетних усилий группы энтузиастов. Главным архитектором на протяжении почти всей работы был Дэн Ингалс (Dan Ingalls), но значительный вклад внесли также Питер Дейч (Peter Deutsh), Гленн Краснер (Glenn Krasner) и Ким МакКолл (Kim McCall). Параллельно, усилиями Джеймса Альтхофа (James Althoff), Роберта Флегала (Robert Flegal), Неда Келера (Ned Kaehler), Дианы Мерри (Diana Merry) и Стива Паца (Steve Putz) разрабатывалась оболочка Smalltalk. Адель Голдберг (Adele Goldberg) и Дэвид Робсон (David Robson) взяли на себя роль летописцев проекта.

Известны пять выпусков языка Smalltalk, обозначаемых по году их появления:

Smalltalk-72, -74. -76, -78, и самое свежее воплощение - Smalltalk-80. Реализации 1972 и 1974 годов заложили основу языка, в частности идею передачи сообщений и полиморфизм, хотя механизм наследования еще не появился. В последующих версиях полноправное гражданство получили классы; этим достигла завершения точка зрения, что все состоит из объектов. Smalltalk-80 был перенесен на многие компьютерные платформы.

Есть также один важный диалект (схожий со Smalltalk-80), получивший название Smalltalk/V. Он создан фирмой Digitalk для IBM PC (Windows и OS/2) и Macintosh. За исключением классов пользовательского интерфейса, библиотеки классов Smalltalk/V в обеих версиях практически идентичны. Среда и инструменты разработки также напоминают Smalltalk-80 [4].

Обзор

Как пишет Ингалс: "Цель проекта Smalltalk - сделать мир информации доступным для детей любого возраста. Вся трудность состоит в том, чтобы найти и применить достаточно простые и эффективные метафоры, которые позволят человеку свободно оперировать самой разнообразной информацией от чисел и текстов до звуковых и зрительных образов" [5]. В основу языка положены две простые идеи:

все является объектами; объекты взаимодействуют, обмениваясь сообщениями.

В табл. А-1 приведены характеристики языка Smalltalk с точки зрения семи основных элементов объектного подхода. Множественное наследование в принципе может быть реализовано за счет переопределения некоторых методов-примитивов [б].

Таблица А-1. Smalltalk

Абстракции	Переменные экземпляра Методы экземпляра Переменные класса Методы класса	Да Да Да Да
Инкапсуляция	Переменных Методов	Закрытые Открытые
Модульность	Разновидности модулей	Нет
Иерархии	Наследование Шаблоны Метаклассы	Одиночное Нет Да
Типизация	Сильная типизация Полиморфизм	Нет Да(одиночный)
Параллельность	Многозадачность	Непрямая (посредством классов)
Сохраняемость	Долгоживущие объекты	Нет

Пример

Рассмотрим задачу со списком разнообразных фигур: окружностей, прямоугольников, закрашенных прямоугольников (см. аналогичную задачу в главе 3). В обширной библиотеке классов Smalltalk уже определены классы для окружности и прямоугольника, поэтому решение задачи упрощается, что подчеркивает роль повторного использования. Однако для целей сравнения предположим, что в нашем распоряжении есть только классы-примитивы для линий и дуг окружности. Определим класс **AShape** следующим образом:

```
Object subclass: №AShape
    instanceVariableNames: 'theCenter'
    classVariableNames: ' ' poolDictionaries: ' '
    category: 'Appendix'
initialize
    "Инициализировать фигуру"
    theCenter:= Point new
setCenter: aPoint
    "Задать центр фигуры"
    theCenter := aPoint
center
    "Вернуть центр фигуры"
    ^theCenter
draw
    "Нарисовать фигуру"
    self subclassResponsibility
```

Теперь определим подкласс **ACircle**:

```
AShane subclass: #ACircle
    instanceVariableNames: 'theRadius'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Appendix'
setRadius: anInteger
"Задать радиус окружности"
```

```
        theRadius := aInteger
radius
"Вернуть радиус окружности"
        ^theRadius
draw
        "Нарисовать окружность"
        | anArc index |
        anArc := Arc new.
        index := 1.
        [index <= 4]
        whileTrue:
        [anArc
        center: theCenter radius: theRadius quadrant: index. anArc display. index :=
index + 1]
```

Далее введем подкласс **ARectangle**:

```
AShape subclass: №ARectangle
        instanceVariableNaines: 'theHeight theWidth'
        classVariableNairies : ' '
        poolDictionaries: ''
        category: 'Appendix'
draw
"Нарисовать прямоугольник"
        | aLine upperLeftCorner |
        aLine := Line new.
        upperLeftCorner := theCenter x - (theWidth    /2)@
        (theCenter y -   (theHeight /2)). aLine beginPoint: upperLeftCorner.
        aLine endPoint: upperLeftCorner x + theWidth  @      upperCorner y.
aLine display.
        aLine beginPoint:: aLine endPoint.
        aLine endPoint: upperLeftCorner x + thewidth   в
        (upperCorner y +        theHeight). aLine display.
        aLine beginPoint: aLine endPoint. aLine endPoint: upperLeftCorner x @
        (upperLeftCorner        y +      theHeight). aLine display.
        aLine beginPoint: aLine endPoint. aLine endPoint: upperLeftCorner. aLine
display
setHeight: anInteger
        "Задать высоту прямоугольника"
        theHeight := anInteger
setWidth: anInteger
        "Задать ширину прямоугольника"
        CheWidth := anInteger
height
        "Вернуть высоту прямоугольника"
        ^theHeight
width
        "Вернуть ширину прямоугольника"
        ^thewidth
```

Наконец, подкласс **ASolidRectangle** опишем так:

```
ARectangle subclass: #ASolldRectangle
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Appendix'
draw
```

"Нарисовать сплошной прямоугольник"

```
I upperLeftCorner lowerRightCorner I
super draw.
upperLeftCorner := theCenter x - (thewidth quo: 2) + 1 в
(theCenter y - (theHeight quo: 2) + 1). lowerRigthCorner :=
upperLeftCorner x + thewidth - 1 в
(upperLeftCorner y + theHeight - 1). Display

fill:   (upperLeftCorner corner: lowerRightCorner)
mask:   Form gray
```

Ссылки

Основными руководствами по языку Smalltalk являются книги "Smalltalk-80:

The Language", Голдберг и Робсон [7]; "Smalltalk-80: The Interactive Programming Environment", Голдберг [8]; "Smalltalk-80: Bit of History Words of Advice", Крас-нер, [9]. ЛаЛонд и Пух [10] подробно исследуют Smalltalk-80, в том числе библиотеки классов и средства разработки приложений.

A.3. Object Pascal

Происхождение

Object Pascal создавался сотрудниками компании Apple Computer (некоторые из которых были участниками проекта Smalltalk) совместно с Никлаусом Виртом (Niklaus Wirth), создателем языка Pascal. Непосредственным предшественником Object Pascal является Clascal (объектно-ориентированная версия Pascal для компьютера Lisa). Object Pascal известен с 1986 года и является первым объектно-ориентированным языком программирования, который был включен в Macintosh Programmer's Workshop (MPW), среду разработки для компьютеров Macintosh фирмы Apple. Для MPW создана библиотека классов, называемая MacApp, являющаяся основой для создания прикладных приложений, отвечающих требованиям к интерфейсу пользователя Macintosh.

Обзор

Шмукер (Schmucker) утверждает, что "Object Pascal - это "скелет" объектно-ориентированного языка. В нем нет методов класса, переменных класса, множественного наследования и метаклассов. Эти механизмы исключены специально, чтобы сделать язык простым для изучения начинающими "объектными" программистами" [II].

В табл. А-2 приведены общие характеристики Object Pascal.

Таблица А-2. Object Pascal

Абстракции	Переменные экземпляра Методы экземпляра Переменные класса Методы класса	Да Да Нет Нет
Инкапсуляция	Переменных Методов	Открытые Открытые
Модульность	Разновидности модулей	Модуль (unit)
Иерархии	Наследование Шаблоны Метаклассы	Одиночное Нет Нет

Типизация	Сильная типизация Полиморфизм	Да Да(одиночный)
Параллельность	Многозадачность	Нет
Сохраняемость	Долгоживущие объекты	Нет

Ссылки

Основным руководством по Object Pascal является "MPW Object Pascal Reference" от Apple [12].1

А.4. C++

Происхождение

Язык программирования С++ был разработан Бьерном Страуструпом, сотрудником AT&T Bell Laboratories. Непосредственным предшественником С++ является С with Classes, созданный тем же автором в 1980 году. Язык С with Classes, в свою очередь, был создан под сильным влиянием С и Simula. С++ - это в значительной степени надстройка над С. В определенном смысле можно назвать С++ улучшенным С, тем С, который обеспечивает контроль типов, перегрузку функций и ряд других удобств. Но главное в том, что С++ добавляет к С объектную ориентированность.

Известны несколько версий С++. В версии 1.0 реализованы основные механизмы объектно-ориентированного программирования, такие как одиночное наследование и полиморфизм, проверка типов и перегрузка функций. В созданной в 1989 году версии 2.0 нашли отражение многие дополнительные свойства (например, множественное наследование), возникшие на базе широкого опыта применения языка многочисленным сообществом пользователей. В версии 3.0 (1990) появились шаблоны (параметризованные классы) и обработка исключений. Комитет ANSI по С++ (X3J16) недавно одобрил предложения по введению пространств имен (что соответствует нашему обозначению категорий классов) и проверки типов во время исполнения.

Первые компиляторы С++ строились на основе препроцессора для языка С, названного cfront. Поскольку этот транслятор создавал промежуточный код на С, он позволил очень быстро перенести С++ практически на все UNIX-системы. Сейчас почти на всех платформах созданы (в том числе коммерческие) "настоящие" компиляторы С++.

Обзор

Страуструп пишет: "С++ создавался с целью избавить автора и его друзей от необходимости программировать на ассемблере, С или других современных языках такого уровня. Главной задачей было придумать язык, на котором удобно писать хорошие программы и с которым программисту приятно работать. С++ никогда не проектировался на бумаге. Его проектирование, документирование и реализация выполнялись одновременно" [13]. С++ исправил многие недостатки С и ввел описания классов, контроль типов, перегрузку функций, управление памятью, постоянные типы, ссылки, встраиваемые функции, производные классы и виртуальные функции [14].

В последние годы этот язык стал очень популярен благодаря системе Delphi фирмы Borland. - Примеч. ред.

Характеристики С++ приведены в табл. А-3.

Таблица А-3. С++

Абстракции	Переменные экземпляра Методы экземпляра Переменные класса Методы класса	Да Да Да Да
Инкапсуляция	Переменных Методов	Открытые, защищенные, закрытые Открытые, защищенные, закрытые
Модульность	Разновидности модулей	файл
Иерархии	Наследование Шаблоны Метаклассы	Множественное Да Нет
Типизация	Сильная типизация Полиморфизм	Да Да(одиночный)
Параллельность	Многозадачность	Непрямая (посредством классов)
Сохраняемость	Долгоживущие объекты	Нет

Пример

Снова вернемся к задаче определения фигур. В C++ принято описывать интерфейсную часть классов в заголовочных файлах. Мы можем написать:

```
struct point {
	int x;
	int y;
	};
class Shape {
public:
Shape();
	void setCenter (Point p};
	virtual void draw() = 0;
	Point center 0 const;
private
	Point theCenter;
};
class Circle : public Shape {
public:
Circle();
	void setRadius(int r);
	virtual void draw();
	int radius() const;
private:
int theRadius;
};

class Rectangle : public Shape   {
public:
Rectangle();
	void setHeight(int h);
	void setWidth(int w);
	virtual void draw();
	int height () conat;
	inc width () const;
```

```
private:
int theHeigh;
        int TheWidth;
};

class SolidRectangle : public Rectangle {
public:
virtual void draw();
};
```

Определение C++ не предполагает наличия библиотеки классов. Для наших целей мы предположим наличие программного интерфейса X Windows и глобальных объектов Display, window, GraphlcsContext (требуемых xlib). Теперь можно завершить разработку, написав в отдельном файле реализацию методов, перечисленных выше:

```
Shape::Shape()
{
        theCenter.x = 0;
        theCenter.y = 0;
};
void Shape::getCenter(Point p)
{
theCenter = p;
};

Point Shape::center() const
{
return theCenter;
};

Circle::Circle() : theRadius(0) {};

void Circle: :setRadius( int r)
{
theRadius = r;
};
void Circle::draw()
{
int x = (center ().x - theRadius);
        int Y = (center().y - theRadius);
        XDrawArc(Display, Window, GraphicsContext, X, Y,
        (theRadius • 2), (theRadius * 2), 0, (360 • 64));
};
int Circle::radius() const
{
return theRadius;

};

Rectangle::Rectangle() : theHeight(0), theWidth(0) {};

void Rectangle: :setHeight( int h)
{
theHeight = h;
};
```

```
void Rectangle: :setWidth( int w)
{
theWidth = w;
};

void Rectangle::draw()
{
int X = (center().x - (theWidth / 2));
        int Y = (center ().y - (theHeight / 2));
        XDrawRectangle (Display, Window, GraphicsContext, X, Y, thewidth,
theHeight);
};

int Rectangle: : height() const
{
return theHeight;
};

int Rectangle: : width () const
{
return thewidth;
};
void SolidRectangle::draw()
{
Rectangle::draw() ;
        int x - (centerO.x - (width() / 2));
        int Y - (center().y - (height() / 2));
        gc oldGraphicsContext = GraphicsContext;
        XSetForeground(Display, OraphicsContext, Gray);
        XDrawFilled(Display, Window, GraphicsContext, X, Y,
        width (), height());
        GraphicaContext = OldGraphicsContext;
};
```

Ссылки

Основной ссылкой по C++ является "Annotated C++ Reference Manual" Эллис и Страуструпа [15]. Кроме того, Страуструп [16] предложил углубленный анализ языка и его использования в контексте объектно-ориентированного проектирования.

A.5. Common Lisp Object System (CLOS)

Происхождение

Существуют буквально десятки диалектов языка Lisp, включая MacLisp, Standard Lisp, SpiceLisp, S-1 Lisp, ZetaLisp, Nil, InterLisp и Scheme. В начале 80-х годов под воздействием идей объектно-ориентированного программирования возникла серия новых диалектов Lisp, многие из которых были ориентированы на представление знаний. Успех в стандартизации Common Lisp стимулировал попытки стандартизировать объектно-ориентированные диалекты в 1986 году.

Идея стандартизации была поддержана летней конференцией ACM по Lisp и функциональному программированию 1986 года, в результате чего

была создана специальная рабочая группа при комитете X3J13 ANSI (комитет по стандартизации Common Lisp). Поскольку новый диалект должен был стать надстройкой над Common Lisp, он получил название Common Lisp Object System (Объектная система Common Lisp) или, сокращенно, - CLOS. Возглавил комитет Дэниел Бобров (Daniel Bobrow), а его членами стали Соня Кин (Sonya Keene), Линда де Мичил (Linda DeMichiel), Патрик Дассад (Patrick Dussud), Ричард Габриель (Richard Gabriel), Джеймс Кемпф (James Kempf), Грегор Кисазлес (Gregor Kicazles) и Дэвид Мун (David Moon).

Серьезное влияние на проект CLOS оказали языки NewFlavors и CommonLoops. После двухлетней работы, в 1988 году была опубликована полная спецификация CLOS.

Обзор

Кип отмечает, что в проекте CLOS ставились три основные цели. CLOS должен:

• представлять собой стандартное расширение языка, включающее все наиболее полезные свойства существующей объектно-ориентированной парадигмы;

• обеспечить эффективный и гибкий интерфейс программиста, позволяющий реализовать большинство прикладных задач;

• проектироваться как расширяемый протокол, так, чтобы можно было изменять его поведение, тем самым стимулируя дальнейшие исследования в области объектно-ориентированного программирования [17].

Обзор характеристик CLOS можно найти и табл. А-4. Не поддерживая непосредственно механизм долгоживущих объектов, CLOS имеет расширения с протоколом метаобъектов, реализующих этот механизм [18].

Таблица А-4. CLOS

Абстракции	Переменные экземпляра Методы экземпляра Переменные класса Методы класса	Да Да Да Да
Инкапсуляция	Переменных Методов	Чтение,запись,доступ Открытые
Модульность	Разновидности модулей	Пакет
Иерархии	Наследование Шаблоны Метаклассы	Множественное Нет Да
Типизация	Сильная типизация Полиморфизм	Возможна Да (множественный)
Параллельность	Многозадачность	Да
Сохраняемость	Долгоживущие объекты	Нет

Ссылки

Основным руководством по языку CLOS является -"Common Lisp Object System Specification" [19].

А.6. Ada

Происхождение

Министерство обороны США, возможно, самый крупный в мире пользователь компьютеров. В середине 70-х годов программные разработки этого департамента достигли критической точки: проекты выходили из

временных и бюджетных рамок, а заданных характеристик достичь не удавалось. Стало очевидно, что дальше ситуация только ухудшится, стоимость разработки программных систем взлетит еще выше, а потребность в программах будет расти экспоненциально. Для решения всех этих проблем, отягощенных вдобавок наличием сотен языков программирования, министерство обороны профинансировало проект создания единого общего языка высокого уровня. В некотором смысле Ada является одним из первых языков программирования, выпущенных промышленным способом. Исходные требования были сформулированы в 1975 году (Steelman) и реализованы в 1978 году. Был объявлен международный конкурс, на который откликнулось 17 участников. Это число затем было сокращено до четырех, затем до двух, и наконец до одного; при этом в проектировании и испытаниях участвовали сотни ученых по всему миру.

Проект-победитель вначале носил условное наименование Green (в конкурсе проект имел зеленый кодовый знак); позднее он получил имя Ada в честь Ады Августы графини Лавлейс (Ada Augusta Lovelace), которая была удостоена этой чести за свои соображения о потенциальных возможностях компьютеров. Основным разработчиком языка был Жан Икбьян (Jean Ichbian) из Франции. В команду разработчиков входили: Бернд Криг-Брюкнер (Bernd Krieg-Brueckner), Бриан Вичман (Brian Wichmann), Анри Ледгар (Henry Ledgard), Жан-Клод Ельяр (Jean-Claude Heliard), Жан-Лу Гайли (Jean-Loup Gailly), Жан-Раймон Абриаль (Jean-Raymond Abrial), Джон Барнс (John Barnes), Майк Вуджер (Mike Woodger), Оливье Рубин (Olivier Roubine), С. А. Шуман (S. A. Schumann) и С. С. Весталь (S. C. Vestal).

Непосредственными предшественниками Ada являются Pascal и его производные, включая Euclid, Lis, Mesa, Modula и Sue. Были использованы некоторые концепции ALGOL-68, Simula, CLU и Alphard. Стандарт ANSI для Ada был окончательно издан в 1983 году. Трансляторы Ada, хотя и не сразу, были реализованы для всех основных архитектур. Будучи созданным благодаря министерству обороны, язык Ada сегодня используется во многих государственных и коммерческих проектах. Ada - традиционный язык разработки больших и сложных систем, например, системы управления воздушным движением в США и Канаде. Стандарты ANSI должны пересматриваться каждые пять лет, поэтому в настоящее время изучается проект Ada 9x. В нем в исходное определение языка внесен ряд незначительных исправлений: уточнения, устранение очевидных пробелов, исправления ошибок. В настоящем виде Ada является объектным, но не объектно-ориентированным языком. Проект 9x подразумевает расширение языка до уровня объектно-ориентированного.

Обзор

Разработчики Ada прежде всего беспокоились о:

- надежности и эксплуатационных качествах программ;
- программировании как разновидности человеческой деятельности;
- эффективности [20].

В табл. А-5 приведены основные характеристики языка Ada с точки зрения объектного подхода.

Таблица А-5. Ada

Абстракции	Переменные экземпляра Методы экземпляра Переменные класса Методы класса	Да Да Нет Нет
Инкапсуляция	Переменных Методов	Открытые,закрытые Открытые,закрытые
Модульность	Разновидности модулей	Пакет

Иерархии	Наследование Шаблоны Метаклассы	Нет(входит в Ada9x) Да Нет
Типизация	Сильная типизация Полиморфизм	Да Нет(входит в Ada9x)
Параллельность	Многозадачность	Да
Сохраняемость	Долгоживущие объекты	Нет

Ссылки

Основным руководством по языку Ada является "Reference Manual for the Ada Programming Language" [21].

A.7. Eiffel

Происхождение

Автор Eiffel Бертран Мейер (Bertrand Meyer) создавал не только язык объектно-ориентированного программирование, но и инструмент проектирования программ.

Несмотря на сильное влияние Simula, Eiffel - вполне самостоятельный объектно-ориентированный язык со своей собственной средой разработки.

Eiffel поддерживает динамическое связывание и статическую типизацию, тем самым обеспечивая гибкость интерфейсов классов в сочетании с безопасным использованием типов. В Eiffel есть несколько важных черт, поддерживающих более жесткий стиль программирования, в том числе параметризованные классы, утверждения и исключения. Мейер считает, что обобщенные классы хорошо дополняют наследование, учитывая горизонтальный уровень общности; новые классы на одном уровне иерархии можно создавать, используя тип в качестве параметра, а не плодя практически одинаковые подклассы.

Неотъемлемой частью языка являются пред- и постусловия, то есть утверждения, которые должны выполняться при входе в метод и выходе из него. Нарушение утверждения вызывает исключительную ситуацию. Ее можно перехватить, обработать и попробовать вызвать тот же метод еще раз.

Обзор

Eiffel поощряет хорошее программирование, добротную спецификацию классов, сильную типизацию и повторное использование, как через наследование, так и через параметризацию. Формальная трактовка исключительных ситуаций позволяет жестко специфицировать интерфейсы классов при реализации.

Eiffel предоставляет законченную среду разработки программ, включая специальный редактор с выделением синтаксиса, генератор документации, библиотеки классов и броузер. Кроме того, поддерживаются средства управления кодом и сборкой программ.

Свойства языка с точки зрения нашей модели показаны в табл. А-6.

Таблица А-6. Eiffel

Абстракции	Переменные экземпляра Методы экземпляра Переменные класса Методы класса	Да Да Нет Нет
Инкапсуляция	Переменных Методов	Закрытые Открытые, закрытые
Модульность	Разновидности модулей	Блок (unit)

Иерархии	Наследование Шаблоны Метаклассы	Множественное Да Нет
Типизация	Сильная типизация Полиморфизм	Да Да
Параллельность	Многозадачность	Нет
Сохраняемость	Долгоживущие объекты	Нет

Ссылки

Лучше всего взять книгу Мейера "Object Oriented Software Construction" [22].

А.8. Другие объектно-ориентированные языки программирования

На рис. А-2 вы найдете названия многих важных объектных и объектно-ориентированных языков, в библиографии есть ссылки на информацию о большинстве из них.

ABCL/1 ABE Acore Act/1 Act/2	Concurrent Smalltalk CSSA CST Director Distributed Smalltalk	Lore Mace MELD Mjolner ModPascal	Plasma 11 POOL-T PROCOL Quick Pascal Quicktalk
EmeraldNew FlavorsSASTAct/3	Eiffel	Neon	ROSSActor
Actors	ExperCommonLisp	NIL	SCOOP
Felix PascalOakUspSelfActra	Extended Smalltalk	0-CPU	SCOOPSAda
Argus	Flavors	Oberon	Simula
ART Berkeley Smalltalk FOOPS ObjectCobot Smalltalk	FOOPlog	Object Assembler	SINA
Beta Blaze	FRL Galileo	Object Lisp Object Logo	Smalltalk AT Smalltalk V
GLISPObject PascalSPOOLC++ Gypsy Objective-C SR C_ talkHybridObjVLispSRLBrouhaha	Garp	Object Oberon	Smallworld C wilh Classes
Cantor	Inheritance	OOPC	STROBE
IntermissionOPALTrellis/OwlClascal	InnovAda	OOPS+	TClassic Ada
KL-OneOrient84/KUniform Common LoopsKRLOTMUNITS Common ObjectsKRSPCOLVulcanCommon ORBITLittle SmalltalkPIEXLISPConcurrent PrologLOOPSPL/LLZoom/VMCLOS	Jasmine	Orbil	Turbo Pascal 5.xCluster 86

Рис. А-2. Объектные и объектно-ориентированные языки программирования

В работе Саундерса [23] дан обзор более восьмидесяти объектно-ориентированных языков. Автор выделил в них семь категория [24]:

- Actor Языки, поддерживающие механизм делегирования.
- Параллельные Объектно-ориентированные, нацеленные на параллелизм.
- Распределенные Объектно-ориентированные языки, нацеленные на обработку распределенных объектов.
- Основанные Языки, поддерживающие теорию фреймов. на фреймах
- Гибридные Объектно-ориентированные надстройки над обычными языкам и.
- Типа Smalltalk Smalltalk и его диалекты.
- Идеологические Приложения объектной ориентированности к другим областям.
- Остальные Все остальные объектные языки.

Словарь терминов

CRC-карточки, CRC cards. CRC — Class/Responsibilities/Collaborators, Класс/Ответствен-ности/Сотрудники; простое, но достаточно эффективное средство мозгового штурма при выявлении ключевых абстракций и механизмов.

абстрактная операция, abstract operation. Объявленная, но не реализованная операция в абстрактном классе. В С++ абстрактные операции объявляются как чисто виртуальные функции-члены.

абстрактный класс, abstract class. Класс, который не может иметь экземпляров. Абстрактный класс пишется в предположении, что его конкретные подклассы дополнят его структуру и поведение, скорее всего, реализовав абстрактные операции.

абстракция, abstraction. Существенные характеристики объекта, которые отличают его от всех других объектов и четко определяют его концептуальные границы для наблюдателя. Абстрагирование — процесс выявления абстракций. Один из основных элементов объектной модели.

агент, **agent.** Объект, который подвергается воздействию со стороны и сам воздействует на другие объекты. Обычно агенты создаются для выполнения некоторой работы по поручению актеров или других агентов.

актер, actor. Объект, воздействующий на другие объекты, но сам не подвергающийся воздействию с их стороны. В некоторых контекстах то же самое, что активный объект.

активный объект, active object. Объект, которому выделен свой поток управления.

алгоритмическая декомпозиция, algorithmic decomposition. Процесс разделения системы на части, каждая из которых отражает этап общего процесса. Применение структурного подхода к проектированию приводит к алгоритмической декомпозиции, которая фокусируется на потоке управления в системе-

архитектура модулей, module architecture. Граф, вершины которого соответствуют модулям, а ребра — отношениям модулей между собой. Архитектура модулей системы представляется совокупностью диаграмм модулей.

архитектура процессов, process architecture. Граф, вершины которого соответствуют процессорам и устройствам, а ребра — соединениям между ними. Для описания архитектуры процессов системы используются диаграммы процессов.

архитектура, architecture. Логическая и физическая структура системы, сформированная всеми стратегическими и тактическими проектными решениями.

ассоциация, association. Отношение, означающее некоторую смысловую связь между классами.

атрибут, attribute. Часть составного объекта (агрегата).

базовый класс, base **class.** Наиболее общий класс в какой-либо структуре классов. В большинстве приложений есть несколько таких корневых классов. В некоторых языках

программирования определяется всеобщий базовый класс, который является суперклассом для всех остальных классов.

блокирующий объект, blocking object. Пассивный объект, способный работать в многопоточном окружении. Вызов операции блокирующего объекта блокирует клиента на все время операции.

видимость, visibility. Способность одной абстракции видеть другую и, таким образом, ссылаться на ее ресурсы извне. Абстракции видимы друг другу, только если они находятся в одном пространстве имен. Контроль экспорта может еще более ограничить доступ к видимым абстракциям.

виртуальная функция, virtual function. Какая-либо операция над объектом. Виртуальная функция может быть переопределена в подклассах, следовательно, ее реализация определяется всем множеством методов, объявленных во всех классах дерева наследования. Термины «обобщенная функция» и «виртуальная функция» взаимозаменяемы.

временная сложность, time **complexity.** Относительное или абсолютное время, за которое выполняется операция.

действие, action. Некое происшествие в системе, требующее, с практической точки зрения, нулевого времени для своего завершения. Действием может быть вызов операции, запуск другого события, начало или остановка деятельности.

делегирование, delegation. При делегировании один объект, ответственный за операцию, передает выполнение этой операции другому объекту.

деструктор, destructor. Операция класса, которая освобождает состояние объекта и/или уничтожает сам объект.

деятельность, activity. Операция, выполнение которой требует некоторого времени.

диаграмма взаимодействий, interaction diagram. Часть системы обозначении объектно-ориентированного проектирования; используется для демонстрации выполнения какого-либо сценария в контексте диаграммы объектов.

диаграмма классов, class diagram. Часть системы обозначений объектно-ориентированного проектирования; используется, чтобы наглядно показать классы и их взаимоотношения в логическом проекте системы. Может представлять всю структуру классов или ее часть.

диаграмма модулей, module diagram. Часть системы обозначений объектно-ориентированного проектирования; используется для демонстрации разбиения классов и объектов по модулям в физическом проекте системы. Диаграмма модулей отображает архитектуру модулей системы.

диаграмма объектов, object diagram. Часть системы обозначений объектно-ориентированного проектирования; используется, чтобы наглядно показать объекты и отношения между ними в логическом проекте системы. Может отражать всю объектную структуру или часть ее; обычно иллюстрирует смысл механизмов в логическом проекте. Отдельная диаграмма объектов — моментальный снимок из жизни системы.

диаграмма переходов и состояний, state transition diagram. Часть обозначений объектно-ориентированного проектирования; используется для отображения пространства состояний данного класса, событий, которые вызывают переход из одного состояния в другое, и действий, возникающих в результате смены состояния.

диаграмма процессов, process diagram. Часть системы обозначений объектно-ориентированного проектирования; используется, чтобы наглядно показать, как процессы размешены по процессорам в физическом проекте системы. Диаграмма процессов отражает архитектуру процессов.

динамическое связывание, dynamic binding. Связывание означает установление соответствия имени (например, объявленной переменной) с классом. Динамическое связывание происходит при выполнении программы в тот момент, когда создается объект, обозначенный именем.

друг, friend. Класс или операция, имеющие доступ к закрытым операциям или данным некоторого класса. Только сам класс может называть своих друзей.

закрытая, private. Часть интерфейса какого-либо класса, объекта или модуля, закрытая (невидимая) для других классов, объектов и модулей.

защищенная, protected. Часть интерфейса какого-либо класса, объекта или модуля, невидимая для всех других классов, объектов и модулей за исключением подклассов.

идентичность, identity. Природа объекта; то, что отличает его от других объектов.

идиома, idiom. Выражение, общепринятое в каком-либо языке программирования или культуре какого-либо приложения, отражающее общепринятый способ использования да-ного языка.

иерархия, hierarchy. Подчинение или упорядочение абстракций. Две типичных иерархии в сложной системе — структура классов (включая иерархию «обшее/частное») и структура объектов (включая иерархию «целое/часть»); иерархии можно также обнаружить в архитектурах модулей и процессов.

инвариант, invariant. Логическое выражение некоторого условия, истинность которого необходимо соблюдать.

инкапсуляция, encapsulation. Процесс разделения элементов абстракции, которые образуют ее структуру и поведение. Служит для отделения внешних обязательств объекта от его реализации.

инстанцирование, instantiation. Подстановка параметров шаблона обобщенного или параметризованного класса; в результате создается конкретный класс, который может иметь экземпляры.

интерфейс, interface. Внешний вид класса, объекта или модуля, выделяющий его существенные черты и не показывающий внутреннего устройства и секретов поведения.

исключение, exception. Возбуждение исключения показывает, что некоторый логический инвариант не соблюдается. В C++ мы возбуждаем исключение, чтобы избежать неправомочное исполнение операций и дать знать о возникшей проблеме другим объектам, которые могут перехватить исключение и принять меры.

использовать, use. Ссылаться на абстракцию извне. итератор, **iterator.** Операция, позволяющая навещать части некоторого объекта.

категория классов, **class category.** Логически полный набор классов, одни из которых видимы для других категорий классов, а другие ~ нет. Классы в категории сотрудничают для предоставления некоторого набора услуг.

класс, **class.** Множество объектов с общей структурой и поведением. Термины «класс» и «тип» в большинстве случаев (но не всегда) взаимозаменяемы. Понятие класса отличается от понятия типа тем, что концентрируется на классификации по структуре и поведению.

класс-контейнер, container class. Класс, экземпляры которого представляют собой коллекции других объектов. Контейнер может быть однородным (коллекции включают экземпляры только одного класса) либо неоднородным (коллекции включают экземпляры разных классов, имеющих обычно общий суперкласс), В C++ контейнеры обычно определяются как параметризованные классы с параметром, обозначающим класс объектов коллекции.

клиент, client. Объект, который пользуется услугами другого объекта либо выполняя операции над последним, либо через доступ к его состоянию,

ключ, key. Атрибут, значение которого однозначно идентифицирует объект.

ключевая абстракция, key abstraction. Класс или объект, являющийся частью словаря предметной области.

конкретный класс, concrete class. Класс, реализация которого завершена и который, поэтому, может иметь экземпляры.

конструктор, constructor. Операция, создающая объект и/или инициализирующая его состояние.

метакласс, metaclass. Класс класса; класс, экземпляры которого сами являются классами.

метод, method. Операция над объектом, определенная как часть описания класса. Не любая операция является методом, но все методы — операции. Термины «метод», «сообщение» и «операция» обычно взаимозаменяемы. В некоторых языках методы существуют сами по себе и могут переопределяться подклассами; в других языках метод не может быть переопределен, — он служит как часть реализации обобщенных или виртуальных функций, которые можно переопределять в подклассах.

механизм, mechanism. Структура, посредством которой объекты сотрудничают друг с другом, осуществляя поведение, которое соответствует требованиям системы.

модификатор, modifier. Операция, изменяющая состояние объекта.

модуль, **module.** Единица кода, служащая строительным блоком физической структуры системы; программный блок, который содержит объявления, выраженные в соответствии с требованиями языка и образующие физическую реализацию части или всех классов и объектов логического проекта системы. Как правило, модуль состоит из интерфейсной части и реализации.

модульность, modularity. Свойство системы, которая была разделена на связные и слабо зацепленные между собой модули.

мономорфизм, monomorphism. Положение теории типов, согласно которому имена (например, переменных) могут обозначать только объекты одного и того же класса.

мощность, cardinality. Число экземпляров класса: число экземпляров, участвующих в связи классов.

наследование, inheritance. Отношение между классами, при котором класс использует структуру или поведение другого (одиночное наследование) или других (множественное наследование) классов. Наследование вводит иерархию «общее/частное» в которой подкласс наследует от одного или нескольких более общих суперклассов. Подклассы обычно дополняют или переопределяют унаследованную структуру и поведение.

обобщенная функция, generic function. Какая-либо операция над объектом. Обобщенная функция класса может быть переопределена в подклассах; следовательно, ее реализация определяется всем множеством методов, объявленных во всех классах дерева наследования. Термины «обобщенная функция» и «виртуальная функция» взаимозаменяемы.

обобщенный класс, generic class. Класс, служащий шаблоном для создания других классов: шаблон параметризуется другими классами, объектами и/или операциями. Обобщенный класс до создания объектов должен быть инстанцирован. Обобщенные классы используются как контейнерные классы. Термины «обобщенный класс» и «параметризованный класс» взаимозаменяемы.

обратный инжиниринг, reverse-engineering. Восстановление логической или физической модели системы по коду. Противопоставляется прямому инжинирингу.

объект, object, Нечто, чем можно оперировать. Объект имеет состояние, поведение и идентичность. Структура и поведение сходных объектов определены в общем для них классе. Термины «экземпляр» и «объект» взаимозаменяемы.

объектная модель, object model. Совокупность основополагающих принципов, лежащих в основе объектно-ориентированного проектирования; парадигма программирования, основанная на принципах абстрагирования, инкапсуляции, модульности, иерархичности, типизации, параллелизма и устойчивости.

объектное программирование, object-based programming. Метод программирования, основанный на представлении программы как совокупности объектов, каждый из которых является экземпляром некоторого типа. Типы образуют иерархию, но не наследственную. В таких программах типы рассматриваются как статические, а объекты имеют более динамическую природу, которую ограничивают статическое связывание и мономорфизм.

объектно-ориентированная декомпозиция, object-oriented decomposition. Процесс разбиения системы на части, соответствующие классам и объектам предметной области. Практическое применение методов объектно-ориентированного проектирования приводит к объектно-ориентированной декомпозиции, при которой мы рассматриваем мир как совокупность объектов, согласованно действующих для обеспечения требуемого поведения.

объектно-ориентированное программирование, object-oriented programming (OOP). Методология реализации, при которой программа организуется, как совокупность сотрудничающих объектов, каждый из которых является экземпляром какого-либо класса, а классы образуют иерархию наследования. При этом классы обычно статичны, а объекты очень динамичны, что поощряется динамическим связыванием и полиморфизмом.

объектно-ориентированное проектирование, object-oriented design (OOD). Методология проектирования, соединяющая процесс объектно-ориентированной декомпозиции и систему обозначений для представления логической и физической, статической и динамической моделей проектируемой системы. Система обозначений состоит из диаграмм классов, объектов, модулей и процессов.

объектно-ориентированный анализ, object-oriented analysis. Метод анализа, согласно которому требования рассматриваются с точки зрения классов и объектов, составляющих словарь предметной области.

объект-член, member object. Часть состояния объекта. В совокупности объекты-члены полностью определяют структуру объекта. Термины «переменная экземпляра», «поле». «объект-член» и «слот» взаимозаменяемы.

ограничение, constraint. Выражение некоторого смыслового условия, которое должно выполняться.

операция класса, class operation. Операция, например, конструктор или деструктор, общая для всего класса и не принадлежащая конкретному объекту.

операция, operation. Нечто, проделываемое одним объектом над другим, чтобы вызвать реакцию. Все операции, которые можно выполнить над каким-либо объектом, сосредоточены в свободных подпрограммах и функциях-членах (методах). Термины «операция», «метод» и «сообщение» взаимозаменяемы.

ответственность, responsibility. Поведение, за которое ответственен объект.

открытая, public. Часть интерфейса какого-либо класса, объекта или модуля, открытая (видимая) для всех классов, объектов и модулей.

параллелизм, concurrency. Свойство, отличающее активные объекты от неактивных.

параллельный объект, concurrent object. Активный объект, способный работать в многопоточной среде.

параметризованный класс, parameterized class. Класс, служащий шаблоном для других классов; шаблон параметризуется другими классами, объектами и/или операциями. Параметризованный класс должен быть инстаццирован до создания объектов. Параметризованные классы используются как контейнеры. Термины «обобщенный класс» и «параметризованный класс» взаимозаменяемы.

пассивный объект, passive object. Объект, не имеющий собственного потока управления.

переменная класса, class variable. Часть состояния класса. Совокупность всех переменных класса образует его структуру. Переменные класса совместно используются всеми его экземплярами. В C++ переменная класса объявляется как статический член.

переменная экземпляра, instance variable. Часть состояния объекта. В совокупности переменные экземпляра полностью определяют структуру объекта. Термины «переменная экземпляра», «поле», «объект-член» и «слот» взаимозаменяемы.

переход, transition. Переход из одного состояния в другое.

поведение, behavior. Действия и реакции объекта, выраженные в терминах передачи сообщений и изменения состояния; видимая извне и воспроизводимая активность объекта.

подкласс, subclass. Класс, наследующий от одного или нескольких классов (которые называются его непосредственными суперклассами).

подсистема, subsystem. Совокупность модулей, часть которых видима для других подсистем, а часть — скрыта.

поле, field. Часть состояния объекта; совокупность полей объекта образуют его структуру. Термины «поле», «переменная экземпляра», «объект-член» и «слот» означают одно и то же.

полиморфизм, polymorphism. Положение теории типов, согласно которому имена (например, переменных) могут обозначать объекты разных (но имеющих общего родителя) классов. Следовательно, любой объект, обозначаемый полиморфным именем, может по-своему реагировать на некий общий набор операций.

последовательное проектирование, round-trip gestalt design. Стиль проектирования, который подчеркивает последовательность и итеративность в развитии системы: посредством уточнения различных, хотя и согласованных логических и физических представлений системы в целом; объектно-ориентированное проектирование основывается на последовательном проектировании, что является выражением взаимозависимости общей картины проекта и его деталей.

последовательный объект, sequential object. Пассивный объект, рассчитанный на работу в однопоточном окружении.

постусловие, postcondition. Инвариант, соблюдаемый на выходе из операции.

поток управления, thread of control. Отдельный процесс. Запуск потока управления приводит к возникновению независимой динамической деятельности в системе; данная система может иметь несколько одновременно выполняемых потоков, некоторые из которых могут динамически возникать и уничтожаться. Многопроцессорные системы допускают истинную многопоточность. в то время как на однопроцессорных компьютерах возможна только иллюзия многопоточности. (Термин «thread of control» переводится также «нить управления». В данном издании принят перевод «поток управления» как более распространенный. Отметим, что в некоторых случаях автор использует термин «flow of control», который переведен также. — *Примеч. ред.)*

предусловие, precondition. Инвариант, предполагаемый на входе в операцию.

примесь, mixin. Класс, реализующий какое-либо четко выделенное поведение; используется для уточнения поведения других классов посредством наследования; поведение примеси обычно ортогонально поведению класса, с которым она смешивается.

пространственная сложность, space complexity, Относительный или абсолютный объем памяти, занимаемый объектом.

пространство состояний, state space. Перечислимое множество всех возможных состояний объекта. Пространство состояний программы содержит неопределенное, но конечное число состояний (не обязательно желаемых или ожидаемых).

протокол, protocol. Способы, которыми объекты могут действовать и реагировать; полное статическое и динамическое представление объекта; протокол объекта определяет допустимое поведение объекта.

процесс, process. Запуск одного потока управления. **процессор, processor.** Часть аппаратного обеспечения, имеющая вычислительные ресурсы.

прямой инжиниринг, forward-engineering. Создание исполнимого кода по логической или физической модели. Противопоставляется обратному инжинирингу.

раздел, partition. Категории классов или подсистемы, составляющие часть данного уровня абстракции.

реактивная система, reactive system. Система, движимая событиями. Поведение такой системы не определяется простым отображением «вход-выход».

реализация, implementation. Внутреннее представление класса, объекта или модуля, включая секреты его поведения.

роль, role. Способность или цель, с которой класс или объект участвует в отношениях с другими; некоторая четко выделяемая черта поведения объекта в определенный момент времени; роль — это лицо, которое объект являет миру в данный момент.

свободная подпрограмма, free subprogram. Процедура или функция, которая выполняется как непримитивная операция над объектом или объектами одного и тоже или различных классов. Свободная подпрограмма — это любая подпрограмма, которая не является методом какого-либо класса.

связь, link. Связь между объектами, экземпляр ассоциации. **селектор, selector.** Операция, имеющая доступ к состоянию объекта, но не изменяющая его.

сервер, server. Объект, который никогда не воздействует на другие объекты, но используется ими; объект, предоставляющий некоторые услуги.

сигнатура, signature. Полная спецификация операции с указанием типов аргументов и возвращаемого значения.

сильно типизированный, strongly typed. Свойство языка программирования, в соответствии с которым во всех выражениях гарантируется согласованность типов.

синхронизация, synchronization. Семантика параллельности операции. Операция может быть простой (присутствует только один поток управления); синхронной (рандеву двух потоков); односторонняя (рандеву, при котором одному из потоков приходится ждать);по истечении времени (рандеву, в котором один процесс ждет другого определенное время); асинхронной (два процесса независимы друг от друга).

система реального времени, real-time system. Система, в которой некоторые существенные процессы должны укладываться в отведенное время. Система «жесткого» реального времени должна быть детерминированной; запаздывание с реакцией грозит катастрофой.

скрытие информации, information hiding. Процесс скрытия всех секретов объекта, которые ничего не добавляют к его существенным характеристикам; обычно скрывают структуру объекта и реализацию его методов.

словарь данных, data dictionary. Полный перечень всех классов в системе. **слой, layer.** Совокупность категорий классов или подсистем одного уровня абстракции.

слот, slot. Часть состояния объекта; совокупность слотов образуют структуру объекта. Термины «поле», «переменная экземпляра», «объект-член» и «слот» означают одно и то же.

событие, event. Что-то, что может изменить состояние системы.

сообщение, message. Операция, которую один объект может выполнять над другим. Термины «сообщение», «метод» и «операция» обычно взаимозаменяемы.

составной объект (агрегат), aggregate object. Объект, состоящий из других объектов (его частей).

состояние, state. Совокупный результат поведения объекта: одно из стабильных условий, в которых объект может существовать, охарактеризованных количественно; в любой конкретный момент времени состояние объекта включает в себя перечень (обычно, статический) свойств объекта и текущие значения (обычно, динамические) этих свойств.

сотрудничество, collaboration. Процесс, в котором несколько объектов сотрудничают для обеспечения требуемого поведения верхнего уровня.

сохраняемость, persistence. Способность объекта существовать во времени, переживая породивший его процесс, и (или) в пространстве, перемещаясь из одного адресного пространства в другое.

среда разработки, framework. Набор классов, предоставляющих некоторые базовые услуги в определенной области. Таким образом, среда разработки экспортирует классы и механизмы, которые клиенты могут использовать или адаптировать.

статическое связывание, static binding. Связывание означает установление соответствия имени (например, объявленной переменной) классу. Статическое связывание происходит при объявлении имени (во время компиляции), до того, как объект будет создан.

страж, **guard.** Логическое выражение, применяемое к событию; если выражение истинно, то событие происходит и система изменяет состояние.

стратегическое проектное решение, **strategic design decision.** Проектные решения, которые имеют решающее влияние на архитектуру.

структура классов, class **structure.** Граф, вершины которого соответствуют классам, а ребра — отношениям классов. Структура классов для конкретной системы представляется в виде совокупности диаграмм классов.

структура объектов, object structure. Граф, вершины которого соответствуют объектам, а ребра — отношениям объектов. Для отражения структуры объектов или ее части используются диаграммы объектов.

структура, **structure.** Конкретное представление состояния объекта. Каждый объект имеет собственное состояние, независимое от других объектов, хотя все объекты одного класса имеют одинаковое представление состояния.

структурное проектирование, structured design. Метод проектирования, основанный на алгоритмической декомпозиции.

суперкласс, **superclass.** Класс, которому наследуют другие классы (называемые непосредственными подклассами).

сценарий, scenario. Последовательность событий, выражающая некий аспект поведения системы.

тактическое проектное решение, tactical design decision. Проектное решение, имеющее ограниченное значение для архитектуры.

тип, type. Определение области допустимых значений, которые может принимать объект, и множества операций, которые могут выполняться над объектом. Термины «класс» и «тип» обычно (но не всегда) взаимозаменяемы; тип отличается от класса тем, что фокусируется на поддержке общего протокола.

типизация, **typing.** Механизмы, препятствующие замене объектов одного типа на другой или, в крайнем случае, жестко ограничивающие такую замену.

трансформационная система, transformational system. Система, поведение которой определяется в терминах отображения «вход-выход» -

управление доступом, access control. Механизм доступа к данным и операциям класса. В С++ открытые элементы доступны *всем,* защищенные элементы доступны подклассам, так называемым друзьям класса и файлам реализации, закрытые элементы доступны реализации и друзьям класса. Наконец, элементы с доступом на уровне реализации доступны только в файле реализации класса.

уровень абстракции, level of abstraction. Относительное упорядочение абстракций по структурам классов, объектов, модулей или процессов. В терминах иерархии «часть/целое» объект находится на более высоком уровне абстракции, чем другие, если он строится на основе этих объектов: в терминах иерархии «общее/частное», высокоуровневые абстракции носят более обобщенный характер, чем низкоуровневые.

услуга, service. Поведение, обеспечиваемое некоторой частью системы.

устройство, device. Часть аппаратуры, не имеющая собственных вычислительных ресурсов.

утверждение, assertion. Логическое выражение некоторого условия, истинность которого необходимо обеспечить.

утилита класса, class utility. Совокупность свободных подпрограмм. На С++ — класс, который состоит только из статических членов и/или функций-членов.

функциональная точка, function point. В контексте анализа требований к системе — отдельное поведение, видимое извне и поддающееся проверке.

функция, **function.** Некоторое преобразование «вход-выход», вытекающее из поведения объекта.

функция-член, member function. Операция над объектом, определенная как часть описания класса. Все функции-члены — операции, но не все операции - функции-члены. Термины «функции-члены» и «методы» взаимозаменяемы. В некоторых языках функции-члены существуют сами по себе и могут переопределяться подклассами; в других языках функция-член не может быть переопределена, — она служит как часть реализации обобщенных или виртуальных функций, которые можно переопределять в подклассах.

экземпляр, instance. Нечто, чем можно оперировать. Экземпляр имеет состояние, поведение и идентичность. Структура и поведение всех экземпляров класса определяются этим классом. Термины «объект» и «экземпляр» взаимозаменяемы.